**Learn ADO.NET by building CRUD Operations in ASP.NET Core**

The **ASP.NET CORE** framework defines a number of namespaces to interact with a **Relational Database System** like Microsoft SQL Server, Oracle, MySQL, etc. Collectively, these namespaces are known as **ADO.NET**.

**Page Contents**

* [ADO.NET – Connected and Disconnected Environments](https://www.yogihosting.com/ado-net-aspnet-core/#environment)
* [ADO.NET Data Providers](https://www.yogihosting.com/ado-net-aspnet-core/#data-providers)
* [Create a Company Database](https://www.yogihosting.com/ado-net-aspnet-core/#database)
* [Create ADO.NET Example Project](https://www.yogihosting.com/ado-net-aspnet-core/#example)
* [Adding Database Connection String](https://www.yogihosting.com/ado-net-aspnet-core/#connection-string)
* [ADO.NET SqlConnection](https://www.yogihosting.com/ado-net-aspnet-core/#sqlconnection)
* [ADO.NET SqlCommand](https://www.yogihosting.com/ado-net-aspnet-core/#sqlcommand)
* [ADO.NET SqlDataReader](https://www.yogihosting.com/ado-net-aspnet-core/#sqlreader)
* [ADO.NET SqlDataAdapter & DataTable](https://www.yogihosting.com/ado-net-aspnet-core/#datatable)
* [Download Source Codes](https://www.yogihosting.com/ado-net-aspnet-core/#download)

In this tutorial, we will learn to work with ADO.NET by communicating with a SQL Server database. First we will **create and open a sql connection to retrieve data**, then perform **inserting, updating, and deletion of data** i.e CRUD Operations.

ADO.NET – Connected and Disconnected Environments

We can use ADO.NET in 2 manners, which are known as environments, these are:

Ezoic

* 1. Connected Environment
* 2. Disconnected Environment

Connected Environment

A **Connected Environment** means the application remains connected with the database throughout the whole length of the operation. Here we typically interact with the database using **connection, command, and data reader** objects.

Disconnected Environment

A **Disconnected Environment** allows data to be retrieved from the data source for being manipulated and later reconciled with the database. Here we use DataTables and DataSets to traverse and manipulate the contents. DataTables and DataSets are client-side copy of external data which is residing in a datasource, we will cover them one by one in the latter half of this tutorial.

Both Connected and Disconnected Environments can be used to Create, Read, Update and Delete data in the data source (database).

ADO.NET Components

What are the ADO.NET components? ADO.NET components perform specific database related tasks. These are 6 main components which are described below:

Ezoic

1. Data Providers – used for communication with the data source.
2. Connection – maintains the location information of the data source.
3. Command – executes a given command type.
4. DataReader – reading data in connected environment.
5. DataAdapter – reading data in disconnected environment.
6. DataTable & DataSet – in-memory storage of data.

We will be seeing each of them in this tutorial.

ADO.NET Data Providers

An **ADO.NET Data Provider** is used for connecting to a database, executing commands and retrieving results. ADO.NET supports multiple data providers, each of which is optimized to interact with a specific database.

Ezoic

The 4 most common Data Providers in ADO.NET are:

* **1. Data Provider for SQL Server** : Provides data access for Microsoft SQL Server Database. It uses System.Data.SqlClient.
* **2. Data Provider for OLE DB** : Provides data access for OLE DB data sources. These are used to access various COM objects and DBMS that does not have a specific .NET data provider. It uses the System.Data.OleDb namespace.
* **3. Data Provider for ODBC** : The ODBC Data Providers are defined within the System.Data.Odbc namespace are typically useful only if we need to communicate with a given DBMS for which there is no custom .NET data provider.
* **4. Data Provider for Oracle** : Provides data access for Oracle databases. Uses the System.Data.OracleClient namespace.

Core objects of ADO.NET Data Providers:

| **Object** | **Description** |
| --- | --- |
| Connection | Used for connecting and disconnecting from the database. The base class of all Connection objects is DbConnection. |
| Command | Executes an SQL Command against a database by using connection and transaction objects. The base class of all Command objects is DbCommand. |
| DataReader | Reads a forward-only, read-only stream of data from a database. The base class for all DataReader objects is the DbDataReader class. |
| DataAdapter | Used to populate a dataset or datatable with the data from the database. The base class is DbDataAdapter. |
| Parameter | Represents a named parameter within a parameterized query. The base class is DbParameter. |
| Transaction | It Encapsulates a database transaction. The base class is DbTransaction. |

Note that the specific names of these base classes will differ among data providers. Example – In the case of Connection objects, SqlConnection versus OdbcConnection, each class derives from the same base class DbConnection.

*It would be correct to assume that after we learn to work with one data provider, the remaining providers work in the same way.*

Important ADO.NET Namespaces:

From a programmatic point of view, the bulk of ADO.NET is represented by a core assembly named  
System.Data.dll.

Ezoic

| **Namespace** | **Description** |
| --- | --- |
| System.Data | This namespace defines the core **ADO.NET** types used by all data providers. It is the lowest common denominator and contains types that are shared among all ADO.NET data providers, regardless of the underlying data store. We cannot build any ADO.NET applications without specifying this namespace in your data access applications. |
| System.Data.SqlClient | This namespace is the **.NET Data Provider for SQL Server**. It gives easy access to the SqlConnection, SqlCommand and other SQL classes. |
| System.Data.Sql | This namespace contains types that allow to discover Microsoft SQL Server instances installed on the current local network. |

There are large number of **ADO.NET data providers**, and in this tutorial we will use the Microsoft SQL Server data provider (System.Data.SqlClient.dll). Recall that this provider allows communicating with the Microsoft SQL Server, including SQL Server Express and LocalDb.

[Entity Framework Core](https://www.yogihosting.com/category/ef-core/) is an Object/Relational Mapping (O/RM) for doing database operations. After completing ADO.NET I would recommend you to learn EF Core also.

If you intend to use ADO.NET to interact with another type of database, you should have no problem doing so once you understand the material presented in the topics that follow.

Create a Company Database

We will create a new database for a company in SQL Server. This database will contain a table called Inventory.

EzoicYou should have SQL Server edition installed in your PC – [Download](https://www.microsoft.com/en-us/evalcenter/evaluate-sql-server-2022)[Link](https://www.microsoft.com/en-us/evalcenter/evaluate-sql-server-2019). SQL Server is a paid version of the database but comes with a free trial. If you are a freebies loving person then use the “SQL Server Express LocalDB” which is totally free to use – [Download SQL Server Express Loc](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver16)[alDB](https://docs.microsoft.com/en-us/sql/database-engine/configure-windows/sql-server-express-localdb?view=sql-server-ver15).

You should also download and install SQL Server Management Studio (SSMS) which is  an integrated environment for managing databases and can be downloaded from – [Download](https://docs.microsoft.com/en-us/sql/ssms/download-sql-server-management-studio-ssms?view=sql-server-ver16).

Connecting to SQL Server

There are 2 ways to connect to SQL Server installed in your local pc. These are through:

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=7423896357875829&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=7423896357875829&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

* 1. SQL Server Object Explorer provided in Visual Studio
* 2. SQL Server Management Studio (SSMS)

Let us now take a look on each of them.

SQL Server Object Explorer

The SQL Server Object Explorer can be opened from the View ➤ SQL Server Object Explorer in Visual Studio.

When SQL Server Object Explorer window opens, right click on the SQL Server node and select Add SQL Server.

Next, a Connect window opens up. In this window, select the Browse tab then click on the Local node to open it.

Inside this node, select the 2nd option that says MSSQLLocalDB. After selecting it, notice the Server Name: field, in the same window, now has the (localdb)\MSSQLLocalDB value.

Now simply click the Connect button to connect to the SQL Server. See the below image:

In a few seconds time the SQL Server will be connected and ready to use.

Ezoic

SQL Server Management Studio (SSMS)

Another way to make a connection with SQL Server is through SQL Server Management Studio (SSMS). First open the SSMS installed in your PC. Then in the Object Explorer window (opened from View ➤ Object Explorer), click the icon that says Connect Object Explorer (opened from File ➤ Connect Object Explorer).

You will see Connect to Server window being opened up. In this window, enter (localdb)\MSSQLLocalDB for the Server name.

Also make sure that Server type is selected as Database Engine, and Authentication is selected as Windows Authentication. This is shown in the below image:

Next, click the Connect button to connect to the SQL Server. The connection will be made in 1 or 2 seconds time, and after this you can create your database in SQL Server.

Company Database

We use SQL Server Object Explorer in Visual Studio to create the database, it’s tables and other things to the SQL Server installed locally. You can also use the SQL Server Management Studio (SSMS) if you wish to, the procedures remain the same.

Move forward by right clicking on Databases node and select Add New Database

In the resulting dialog, enter Company as the database name and click OK to create the database.

The database is created but is empty. So now we will first create the Inventory table.

Important tutorial – [How to Perform CRUD Operations in SQL Server](https://www.yogihosting.com/crud-operations-sql-server/)

Click the ► sign given in front of Company database name to open it.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=1686462351849442&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

Next, right click on the Tables node and select Add New Table.

A Table Design Window will open. Here we can add the fields for the Inventory table.

In this window, inside the T-SQL tab (see image below), we can enter the below given sql script to create the ‘Inventory’ table:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | CREATE TABLE [dbo].[Inventory]  (      [Id] INT NOT NULL IDENTITY(1,1) PRIMARY KEY,      [Name] VARCHAR(50) NOT NULL,      [Price] MONEY NOT NULL,      [Quantity] INT NOT NULL,      [AddedOn] DATE NOT NULL DEFAULT GETDATE()  ) |

Next, click the Update button to start the creation process, see the below image which explains this:

We will get a new dialog box called Preview Database Updates. Here click the Update Database button and within a few seconds the Inventory table will be created.

Understanding the Inventory Table Script

In the Inventory table there are 5 columns which are Id, Name, Price, Quantity, & AddedOn. All these columns are provided with their data types just next to them. Like INT for Id column, VARCHAR(50) for the Name column and so on. These are listed below.

Ezoic

INT --> for integer values like 1,2,100

VARCHAR(50) --> for strings up to 50 characters in length

MONEY --> for numeric values of 4 decimal place like 20.5000, 5.4999

DATE --> for date values like 11-20-2018

All these 5 columns are also forced to contain at least some value of their given type. This is done by using the NOT NULL keyword.

Now see the Id column definition which is:

|  |  |
| --- | --- |
| 1 | [Id] INT NOT NULL IDENTITY(1,1) PRIMARY KEY |

We have stated it to be a primary key by using the PRIMARY KEY keyword. That means all Ids will be unique and there can be no two Ids in the table with the same value.

The keyword IDENTITY(1,1) is used to perform an auto-increment feature. So the first Id column’s value will be 1, and it will be increment by 1 for each new record.

Ezoic

Now see the AddedOn column whose definition is given below:

|  |  |
| --- | --- |
| 1 | [AddedOn] DATE NOT NULL DEFAULT GETDATE() |

It is given a keyword DEFAULT GETDATE() to specify current system date in YYYY-MM-DD format, and this value will be automatically added to this column.

Create ADO.NET Example Project

We are now ready to create **ADO.NET** Codes that works with the database we just created. So first create a new project in Visual Studio by selecting the ASP.NET Core Web App (Model-View-Controller) template.

Name this project as ADO. We are using the Dot Net 6.0 version, you can read [First ASP.NET Core 6.0 MVC Application](https://www.yogihosting.com/aspnet-core-first-application/) where all the creation steps are explained one by one.

With the project created, we need to add the MVC components. Follow the below section to do these.

Model

To the Models folder on the root of the project, add a class called Inventory.cs whose code is given below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **namespace** ADO.Models  {  **public** **class** Inventory      {  **public** **int** Id { **get**; **set**; }  **public** **string** Name { **get**; **set**; }  **public** **decimal** Price { **get**; **set**; }  **public** **int** Quantity { **get**; **set**; }  **public** DateTime AddedOn { **get**; **set**; }      }  } |

The class contains properties that are copies of the fields of the Inventory table. These properties will contains the records values read by ADO.NET from the database.

Controller

We are going to use the HomeController given inside the Controllers folder to create ADO.NET features. The code of the HomeController should look as shown below.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **using** Microsoft.AspNetCore.Mvc;    **namespace** ADO.Controllers  {  **public** **class** HomeController : Controller      {  **public** IActionResult Index()          {  **return** View();          }      }  } |

Better Responsive designs are created with Bootstrap. Install Bootstrap to your projects and start using it. I have written a separate tutorial called [How to Install Bootstrap Package in ASP.NET Core Application in Visual Studio](https://www.yogihosting.com/install-bootstrap-aspnet-core/).

Install System.Data.SqlClient Package

We will have to install System.Data.SqlClient package through NuGet in our project. The System.Data.SqlClient is the .NET Data Provider for SQL Server and through it we will be able to communicate with the database.

Adding Database Connection String

A **Database Connection String** is a string that stores the information required to connect the application to the database. This information is the name of the driver, Server name and Database name. Connection string also contains security information such as user name and password needed for making a secured connection to the database.

Ezoic

In ADO.NET the connection string is stored inside appsettings.json file of ASP.NET Core project. This file resides on the root of the project.

If your project don’t have it the you need to add it to your project. To add appsettings.json file, right click on the project name in the Solution Explorer and select Add ➤ New Item. Then in the Add New Item dialog, select App Settings File, this file will have the name as appsettings.json (see below image). Finally click the Add button to add it to the project.

The file will be created and opens up in Visual Studio. Delete everything from it and add the below given connection string to it, and finally save the file:

Ezoic

{

"ConnectionStrings": {

"DefaultConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Company;Trusted\_Connection=True;MultipleActiveResultSets=true"

}

}

Note that we gave the connection string value inside the ConnectionStrings > DefaultConnection node. The following informations are specified:

* Server – (localdb)\\MSSQLLocalDB. It specifies the name of the SQL Server instance the application will be connecting to.
* Database – Company. It specifies the name of the database contained by the SQL Server instance installed in our system.
* Trusted\_Connection – True. It specifies that the Windows credentials of the current logged in user are used to authenticate against the SQL Server.
* MultipleActiveResultSets – true. It is an optional parameter and you can remove it from the connection string. Multiple Active Result Sets (MARS) is a feature that works with SQL Server to allow the execution of multiple batches on a single connection. It’s a sort of performance enhancing technique.

*If you don’t want to use Trusted\_Connection=true, you need to specify user id and password (of the SQL Server instance) explicitly in the connection string, and leave out any reference to Trusted\_Connection. This is used in production scenario when we make the website live on the internet.*

An example of the connection string in the production environment is given below.

{

"ConnectionStrings": {

"DefaultConnection": "Data Source=189.128.1.130,1440;Initial Catalog=MovieDB;Persist Security Info=True;User ID=SA;Password=vV5r9tn0M4@"

}

}

Reading Connection String in Controller

When we perform CRUD Operations in the Controller we need to read the connection string from appsettings.json file. This is done by adding dependency of IConfiguration type in the controller’s constructor. The Dependency Injection feature of ASP.NET Core will resolve this dependency and then we can easily read the connection string value in the controller.

Dependency for IConfiguration is added by adding the below code to the controller.

public IConfiguration Configuration { get; }

public HomeController(IConfiguration configuration)

{

Configuration = configuration;

}

Now we can read the connection string as shown by the below code:

string connectionString = Configuration["ConnectionStrings:DefaultConnection"];

Proceed by updating the HomeController class to include the necessary codes for reading the connection string. See the highlighted code shown below.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=4690664043895664&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20 | **using** Microsoft.AspNetCore.Mvc;    **namespace** ADO.Controllers  {  **public** **class** HomeController : Controller      {  **public** IConfiguration Configuration { **get**; }  **public** HomeController(IConfiguration configuration)          {              Configuration = configuration;          }    **public** IActionResult Index()          {  **string** connectionString = Configuration["ConnectionStrings:DefaultConnection"];    **return** View();          }      }  } |

ADO.NET SqlConnection

**ADO.NET SqlConnection** is a class from the System.Data.SqlClient namespace and is used to make a connection with the database. It is equivalent to a network connection we make to view movies on NetFlix. SqlConnection is used together with SqlDataAdapter and SqlCommand to perform CRUD Operations and other database related stuffs.

In the below code line we are creating SqlConnection object by providing the database connection string to it’s parameter.

SqlConnection connection = new SqlConnection("Server=(localdb)\\MSSQLLocalDB;Database=Company;Trusted\_Connection=True;MultipleActiveResultSets=true");

//or

string connectionString = Configuration["ConnectionStrings:DefaultConnection"];

SqlConnection connection = new SqlConnection(connectionString);

ADO.NET SqlCommand

**ADO.NET SqlCommand** of System.Data.SqlClient namespace derives from the DbCommand parent class. It is used to **submit and run SQL Queries** to the database. We can specify the type of the command using the CommandType property, which can take any 3 values from the CommandType enum.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=5886972051915140&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | **public** **enum** CommandType  {      StoredProcedure,      TableDirect,      Text // Default value.  } |

In the **SqlCommand** object, we can provide 2 parameters. First for the SQL query and second for the connection object to use.

Consider the code snippet where we have created a SqlCommand object by giving the SQL Query and the connection string to it’s constructor:

|  |  |
| --- | --- |
| 1  2  3  4 | **string** connectionString = Configuration["ConnectionStrings:DefaultConnection"];  SqlConnection connection = **new** SqlConnection(connectionString);  **string** sql = "Select \* From Inventory";  SqlCommand command = **new** SqlCommand(sql, connection); |

Kindly note that, at this point, we have not literally submitted/executed the SQL query to the database but instead prepared the state of the SqlCommand object for future use.

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=7924272537902108&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

There are 3 methods of **SqlCommand** that submit and run the SQL Queries on the database. All of these 3 methods need an open state of the **SqlConnection object** in order to execute the query. These methods are listed in the below table:

| **Name** | **Description** |
| --- | --- |
| ExecuteReader() | Executes (runs) a SQL query and returns the data provider’s SqlDataReader object, which provides forward-only, read-only access for the result of the query. |
| ExecuteNonQuery() | Executes a SQL nonquery (e.g., insert, update, delete, or create table). |
| ExecuteScalar() | A lightweight version of the ExecuteReader() method that was designed specifically for singleton queries (e.g., obtaining a record count). |

In this code snippet we are executing an SQL query on the database using the ExecuteNonQuery method.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | **string** connectionString = Configuration["ConnectionStrings:DefaultConnection"];  SqlConnection connection = **new** SqlConnection(connectionString);  **string** sql = "Select \* From Inventory";  SqlCommand command = **new** SqlCommand(sql, connection);    //opening connection and executing the query  connection.Open();  command.ExecuteNonQuery();  connection.Close(); |

There are also other important members of SqlCommand object that are listed in the below table:

Ezoic

| **Name** | **Description** |
| --- | --- |
| CommandTimeout | A property to ‘get or set’ the time to wait while executing the SqlCommand before terminating the attempt and generating an error. The default value is 30 seconds. |
| Connection | A property to get or set the SqlConnection used by the SqlCommand. |
| Parameters | A property to get the collection of SqlParameter objects used for a parameterized query. |
| Cancel | A method to cancel the execution of the SqlCommand. |

Now you are ready to use SqlCommand class to insert records into the database table.

ADO.NET SqlDataReader

The **ADO.NET SqlDataReader** of the System.Data.SqlClient namespace is used to read the data stored in a database table. The **SqlDataReader class** derives from the parent DbDataReader class.

**SqlDataReader** is mainly used to iterate over large amounts of data quickly and without maintaining an in-memory representation of this data. Be aware, that **SqlDataReader maintains an open connection to their database** until the connection is explicity closed.

Ezoic

We obtain SqlDataReader object from the SqlCommand object by making a call to ExecuteReader() method. The below snipped shown this by reading data from the Inventory table.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | **string** connectionString = Configuration["ConnectionStrings:DefaultConnection"];  SqlConnection connection = **new** SqlConnection(connectionString);    **string** sql = "Select \* From Inventory";  SqlCommand command = **new** SqlCommand(sql, connection);    connection.Open();    // Obtain a data reader via ExecuteReader().  **using** (SqlDataReader dataReader = command.ExecuteReader())  {      // Loop over the results  **while** (dataReader.Read())      {  **int** inventoryId= Convert.ToInt32(dataReader["Id"]);  **string** inventoryName = Convert.ToString(dataReader["Name"]);  **string** inventoryPrice = Convert.ToDecimal(dataReader["Price"]);  **int** inventoryQuantity = Convert.ToInt32(dataReader["Quantity"]);          DateTime inventoryAddedOn = Convert.ToDateTime(dataReader["AddedOn"]);      }  }    connection.Close(); |

The Read() method advances the SqlDataReader to the next record.

Remove Hard Coded Column Names

Notice in the above code we have hard coded the column names in order to get their values, like:

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=1918184939855746&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

dataReader["Name"]

dataReader["Price"]

dataReader["Quanity"]

This can be avoided by using the FieldCount property along with GetName() and GetValue() methods of the SqlDataReader class. These are explained below:

* **FieldCount** – returns the total number of columns in the current record of the SqlDataReader object.
* **GetName()** – gets the name of the specified column.
* **GetValue()** – gets the value of the specified column.

The below code shows how to remove the hard-coded columns names:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | **using** (SqlDataReader dataReader = command.ExecuteReader())  {  **while** (dataReader.Read())      {  **for** (**int** i = 0; i < dataReader.FieldCount; i++)          {  **string** currentColName = dataReader.GetName(i);  **string** currentColValue = Convert.ToString(dataReader.GetValue(i));          }      }  } |

Obtaining Multiple Result Sets from SqlDataReader

We can obtain results from 2 database tables by running 2 SQL queries like:

string sql = "Select \* From Inventory; Select \* from Report";

These form the Multiple Result Sets. We can iterate over each of these results using the SqlDataReader class’s NextResult method. In the below code we iterated over all the rows of every Result Set.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12 | **do**  {  **while** (dataReader.Read())      {  **for** (**int** i = 0; i < dataReader.FieldCount; i++)          {  **string** currentColName = dataReader.GetName(i);  **string** currentColValue = Convert.ToString(dataReader.GetValue(i));            }      }  } **while** (dataReader.NextResult()); |

ADO.NET SqlDataAdapter and DataTable

We can also use ADO.NET SqlDataAdapter class to **read data from the database table**. But unlike SqlDataReader, the SqlDataAdapter object opens the connection automatically when filling the data from the database, it also closes the connection as soon as the data is fetched completely. Therefore we say that it works in the disconnected environment.

When defining an **SqlDataAdapter object** provide it’s constructor with the **SqlCommand object**, like this:

**[×](https://go.ezodn.com/ads/charity/proxy?p_id=1560a79e-1a1d-4fd3-49c5-4a8be17bd249&d_id=122531&imp_id=2891099193841139&c_id=1134&l_id=10016&url=https%3A%2F%2Fjoinourvillage.org%2Fdonate%2F&ffid=1&co=IN)**

Ezoic

SqlDataAdapter dataAdapter = new SqlDataAdapter(sqlcommand);

The SqlDataAdapter has a fill() method to add the records from the database table to a DataTable object.

*ADO.NET DataTable is a single table of in-memory data which is populated from a data source such as Microsoft SQL Server using a DataAdapter. This data of the DataTable is local to the Dot Net application in which it resides. DataTable resides inside the System.Data namespace. We also have a ADO.NET DataSet which is made up of a collection of DataTables. It also contains relationships, and constraints among tables.*

So that means after filling, the DataTable has all the records in the memory, and we can iterate over the records as many times as we need, or even look up a specific record by it’s index. The below code snippet shows how the SqlDataAdapter is filling records to a DataTable object.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **string** connectionString = Configuration["ConnectionStrings:DefaultConnection"];  **using** (SqlConnection connection = **new** SqlConnection(connectionString))  {      DataTable dataTable = **new** DataTable();    **string** sql = "Select \* From Inventory";      SqlCommand command = **new** SqlCommand(sql, connection);        SqlDataAdapter dataAdapter = **new** SqlDataAdapter(command);        // filling records to DataTable      dataAdapter.Fill(dataTable);  } |

If we want to read records using a Stored Procedure then the above code becomes:

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | **string** connectionString = Configuration["ConnectionStrings:DefaultConnection"];  **using** (SqlConnection connection = **new** SqlConnection(connectionString))  {      DataTable dataTable = **new** DataTable();    **string** sql = "ReadInventory"; //ReadInventory is the stored procedure name      SqlCommand command = **new** SqlCommand(sql, connection);      command.CommandType = CommandType.StoredProcedure;        SqlDataAdapter dataAdapter = **new** SqlDataAdapter(command);        dataAdapter.Fill(dataTable);  } |