PROJECT DOCUMENT

**1.ADD TRANSIENT**  
 When using AddTransient , a new instance of the service is created every time it's requested, and it's disposed of when the request is finished.

* This is suitable for services that are stateless and don't maintain any state between requests.
* In this always create a new instance when they are requested in same scoped or other.

Syntax:

services.AddTransient<ITableService, TableService>();

**2.** **AddScoped**

* When you use AddScoped, a single instance of the service is created for each scope. In a web application, a scope typically corresponds to an HTTP request.
* The service instance is reused throughout the scope's lifetime and then disposed of when the scope ends.
* his is useful for services that maintain some state during a single HTTP request.
* In this first and second instance are the same when the call HTTP request it will change .

Syntax:  
services.AddScoped<IWaiterService, WaiterService>()

**3.** AddSingleton

* AddSingleton is like having a single head chef for the entire restaurant. Regardless of how many customers come and go, there's only one head chef.
* a single instance of the service is created for the lifetime of the application.
* This means the service is shared among all requests and clients throughout the lifetime of the application.
* In this each instance are the same when they are call HTTP Request first and second

Configure this services in a startup.cs:

public void ConfigureServices(IServiceCollection services) {

// Transient: New waiter for each table (per request)

services.AddTransient<ITableService, TableService>();

// Scoped: One waiter per group of customers (per request)

services.AddScoped<IWaiterService, WaiterService>();

// Singleton: One head chef for the entire restaurant (shared)

services.AddSingleton<IChefService, ChefService>();

}

Connecting to a database

Using(): This statement is used to defines a scope at the end of which an object is disposed.

This is used to handle the memory allocations and also it disposed the resource code when its closed the execution.

Syntax: key is used to handle the disposal of the resource code.

using(SqlConnection conn = new SqlConnection()) {  
 conn.Connectionstrings="Server=[server\_name];Database=[database\_name];Trusted\_Connection=true";  
conn.Open();

// using code here..  
conn.Close();

}

Conn.Open(): This method is used to open new connection to sql server and you can execute the commands again your data.  
conn.Close():This method is used to close the connection object,record object,recode dataset object and stream of object;   
when connection is closed, it will not remove from the memory.  
  
ConnectionStrings: this is provide the name of the server and name of the database to connect.

DataTable : it’s a ADO.NET class is used for representation of database table. It’s having table rows and columns and create new DataTable or Add data to DataTable and bind the data from the DataTable to Grid View. Its represents a single data table.

Syntax:

SqlDataAdapter da = new SqlDataAdapter();

DataTable dt = new DataTable();  
da.Fill(dt);

SqlDataAdapter: This is used to fill the dataset.

Syntax: SqlDataAdapter adapter = new SqlDataAdapter();

Adapter.Fill(datatable);

SqlCommand: This is use to performance of Create/Read/Insert/Delete/Update operations and execute the stored procedures.

Syntax:

String sqlQuery ="SELECT \* FROM TABLE" or "StroedProcedureName"

SqlCommand cmd = new SqlCommand(sqlQuery,connection)

SqlDataReader: This class is used to read the data from the sql server database most efficient manner. Its reads the data forward direction only. Once it reads the record never go back to previous record.

Syntax: SqlDataReader reader = command.ExecuteReader()

// Call Read before accessing data.

while (reader.Read()) {

string name = reader["Name"].ToString();

string age = reader["Age"].ToString();

string city = reader["City"].ToString();

}

// Call Close when done reading.

reader.Close();

# SqlDataReader.Close Method:

Close Method is used to fill values for the output parameters. Its returns the values and Records are affected on the SqlDataReader by consuming any pending records. After reading is done we have to dispose or block the ExecuteReader by using following statement.

Syntax: SqlDataReader reader = command.ExecuteReader()

// Call Read before accessing data.

while (reader.Read()) {

string name = reader["Name"].ToString();

}

// Call Close when done reading.

reader.Close();

ExecuteReader(): This method used to sends CommandText to connections and also execute the sql commands or stored procedures and returns data set rows from the database.

Syntax: SqlDataReader reader = command.ExecuteReader()

// Call Read before accessing data.

while (reader.Read()) {

string name = reader["Name"].ToString();

string age = reader["Age"].ToString();

string city = reader["City"].ToString();

}

// Call Close when done reading.

reader.Close();

ExecuteNonQuery():This method is used to execute SQL commands and stored procedure performs INSERT, DELETE, UPDATE, GET and It does not return any data from the database and its returns only integers value any number of rows inserted or deleted or updated.

Syntax:

Using(SqlConnection con = new SqlConnection()){

con.Open();

Sqlcommand cmd= new SqlCommand(PROC\_Name,con);  
 Cmnd.CommandType = CommandType.StoredProcedure;

Cmnd.Parameters.AddWithValue("@ID", SqlDbType.Int).Value = Id;

Cmnd.Parameters.AddWithValue("@NAME", SqlDbType.NVarChar).Value = Name;

int result = cmd.ExecuteNonQuery(); // 1 or 0

con.Close();

}

Executescalar(): This is used to execute SQL Commands and Stored procedures after executing returning a single value from the database

and also its returns first column of the first row from the result set from the database.

Syntax:

Using(SqlConnection con = new SqlConnection()){

con.Open();

Sqlcommand cmd= new SqlCommand("SELECT NAME FROM TABLE\_NAME WHERE ID=@ID",con);  
 Cmnd.CommandType = CommandType.StoredProcedure;

Cmnd.Parameters.AddWithValue("@ID",Id);

Cmnd.Parameters.AddWithValue("@NAME ,Name);

String Name = (string)cmd.ExecuteScalar()

con.Close();

}

DataSet: Its represents multiple Tables as Datatable Objects. It has Dis-connected Architecture. In this case SqlDataAdapter class is used to fill the data. It does not require Open or active the connection to database. It stores the one or more data table records. Its a collection of data tables contains data.

Example:

CREATE PROC DATASET

AS BEGIN

Select \* from students; / table1

Select \* from Employees; /table2

END

Syntax:

SqlDataAdapter da = new SqlDataAdapter(DATASET,con);

DataSet ds = new DataSet(); // fetch table1 table2 data

da.Fill(ds);

foreach(DataRow row in ds.Table[0].Rows){

Console.WriteLine(row[0]++row[1]++row[3]++row[4])

}

foreach(DataRow row in ds.Table[1].Rows){

Console.WriteLine(row[0]++row[1]++row[3])

}

// with table name

ds.Tables[0].TableName = "Students";

ds.Tables[2].TableName = "Employees";

foreach(DataRow row in ds.Table["Students"].Rows){

Console.WriteLine(row[0]++row[1]++row[3])

}

foreach(DataRow row in ds.Table["Employee"].Rows){

Console.WriteLine(row[0]++row[1]++row[3])

}

With: SqlQUERY

string Stuquery = "SELECT \* FROM Students";

string EmpQuery = "SELECT \* FROM Employee";

SqlDataAdapter student = new SqlDataAdapter(Stuquery,con);

SqlDataAdapter employee = new SqlDataAdapter(EmpQuery,con);

DataTable studt = new DataTable();

DataTable empdt = new DataTable();

student.Fill(studt);

employee.Fill(empdt);

DataSet ds = new DataSet()

ds.Tables.Add(studt);

ds.Tables.Add(empdt);

foreach(DataRow row in ds.Table[0].Rows){

Console.WriteLine(row[0]++row[1]++row[3]++row[4])

}

foreach(DataRow row in ds.Table[1].Rows){

Console.WriteLine(row[0]++row[1]++row[3])

}

Async : Async keyword turns a method into a async method and which is always allows a await keyword inside a body. This async method allows code to be executed concurrently without blocking the execution of the calling thread. In other words, asynchronous code can run in the background while other code is executing.

 An async keyword is a method that performs asynchronous tasks such as fetching data from a database, reading a file, etc, they can be marked as “async”.

Syntax:

public async Task Get(){ // method with async keyword

var message ="Hellow",  
 return await Task.FromResult(message);

}

Await: Whereas await keyword making  “await” to a statement means suspending the execution of the async method it is residing in until the asynchronous task completes. After suspension, the control goes back to the caller method. Once the task completes, the control comes back to the states where await is mentioned and executes the remaining statements in the enclosing method.

Syntax:

public async Task Get(){

var message ="Hellow",  
 return await Task.FromResult(message); // return statement with await keyword

}  
Task:

Task is a higher-level abstraction for running code asynchronously. A Task denotes a unit of work that needs to be executed asynchronously, and it may or may not return a value. A Task is usually created with the help of the Task Factory class, which provides several methods for creating and executing Tasks.

Syntax:

Public async Task Get(){ // method with task keyword

// code….  
}

Task.FromResult in Async Methods: It is a useful method in async programming to create a completed Task with a **specific result**. It also allows us to quickly return a completed Task in situations where we don’t have asynchronous operations.

Syntax:

public async Task Get(){

var message ="Hellow",  
 return await Task.FromResult(message);

}

Task.CompletedTask: This method returns a task without actually performing any asynchronous work.

Syntax:

public Task UseTaskCompleted(){  
 Console.WriteLine("Hellow");

return Task.CompletedTask;  
 }