Custom ADO.NET with sqlHelpers methods in Data Access Layer

To create a common sqlHelpers methods.  
SqlHelpers.cs:

using Dapper;

using Microsoft.AspNetCore.Mvc.TagHelpers.Cache;

using System.Data;

using System.Data.SqlClient;

using System.Runtime.InteropServices;

namespace JWTRoleAuthentication.CommonLayer.Models

{

//public class SqlHelpers : IDisposable

public class SqlHelpers : IAsyncDisposable

{

private readonly string \_connectionString;

private SqlConnection \_connection;

public SqlHelpers(string connectionString)

{

\_connectionString = connectionString;

\_connection = new SqlConnection(connectionString);

}

//public void Dispose()

//{

// if (\_connection != null)

// {

// if(\_connection.State != System.Data.ConnectionState.Closed)

// {

// \_connection.Close();

// }

// \_connection.Dispose();

// \_connection = null;

// }

//}

public async ValueTask DisposeAsync()

{

if (\_connection != null)

{

if (\_connection.State != System.Data.ConnectionState.Closed)

{

await \_connection.CloseAsync();

}

\_connection.Dispose();

\_connection = null;

}

}

public async Task EnsureConnection()

{

if (\_connection != null)

{

if(\_connection.State != ConnectionState.Open)

{

await \_connection.OpenAsync();

}

}

}

// sql command:

public SqlCommand CreateCommand(string commandText, CommandType commandType = CommandType.StoredProcedure)

{

var command = \_connection.CreateCommand();

command.CommandText = commandText;

command.CommandType = commandType;

return command;

}

// Create StoredProcedure command:

//public SqlCommand CreateStoredProcedureCommand(string storedProcedureName)

//{

// return CreateCommand(storedProcedureName,CommandType.StoredProcedure);

//}

public SqlCommand CreateStoredProcedureCommand(string storedProcedureName)

{

//var command = \_connection.CreateCommand();

//command.CommandText = storedProcedureName;

//command.CommandType = CommandType.StoredProcedure;

//return command;

var command = new SqlCommand(storedProcedureName, \_connection);

command.CommandType = CommandType.StoredProcedure;

return command;

}

// SqlDataAdapter and SqlDataReader

// To Read Table Data using SqlDataAdapter and SqlDataReader for

public async Task<DataTable> ExecuteDataTableAsync(SqlCommand command)

{

DataTable dataTable = new DataTable();

await EnsureConnection();

using(var adapter = new SqlDataAdapter(command))

{

await Task.Run(() => adapter.Fill(dataTable));

return dataTable;

}

}

public async Task<DataTable> ReadDataTable(SqlCommand command)

{

DataTable dataTable = new DataTable();

await EnsureConnection();

using(var reader = await ExecuteReader(command))

{

dataTable.Load(reader);

return dataTable;

}

}

public async Task<DataSet> GetMultipleTablesData(string[] storedProcNames, SqlParameter[] parameters = null)

{

await EnsureConnection();

var dataSet = new DataSet();

var tasks = storedProcNames.Select(async procName =>

{

using (var cmd = CreateStoredProcedureCommand(procName))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

using (var adapter = new SqlDataAdapter(cmd))

{

var dataTable = new DataTable();

await Task.Run(() => adapter.Fill(dataTable));

dataSet.Tables.Add(dataTable);

}

}

});

await Task.WhenAll(tasks);

return dataSet;

}

// To Read database records using SqlDataReader: for read single row and multiple rows and datarowcollections.

// use following methods:

public async Task<SqlDataReader> ExecuteReader(SqlCommand command)

{

// await EnsureConnection();

//return await command.ExecuteReaderAsync(CommandBehavior.CloseConnection);

return await command.ExecuteReaderAsync();

}

public async Task<DataRow> GetSingleRow(string storedProcedureName, params SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcedureName))

{

command.Parameters.AddRange(parameters);

using (var reader = await ExecuteReader(command))

{

var dataTable = new DataTable();

dataTable.Load(reader);

if (dataTable.Rows.Count > 0)

{

return dataTable.Rows[0];

}

}

}

return null;

}

public async Task<List<DataRow>> GetMultipleRows(string storedProcedureName, SqlParameter[] parameters)

{

await EnsureConnection();

var result = new List<DataRow>();

using (var command = CreateStoredProcedureCommand(storedProcedureName))

{

if(parameters != null)

{

command.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(command))

{

var dataTable = new DataTable();

dataTable.Load(reader);

if(dataTable.Rows.Count > 0)

{

result = dataTable.AsEnumerable().Select((row) => row).ToList();

}

}

}

return result;

}

public async Task<DataRowCollection> GetMultipleDataRows(string storedProcedureName, SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcedureName))

{

if (parameters != null)

{

command.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(command))

{

var dataTable = new DataTable();

dataTable.Load(reader);

if(dataTable.Rows.Count > 0)

{

return dataTable.Rows;

}

}

}

return null;

}

// ExecuteNonQuery: for Update and Insert Data Records

// using following methods:

public async Task<int> ExecuteNonQueryAsync(SqlCommand command)

{

await EnsureConnection();

return await command.ExecuteNonQueryAsync();

}

public async Task<int> InsertTable(string storedProcName,params SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcName))

{

if(parameters != null)

{

command.Parameters.AddRange(parameters);

}

int rowAffected = await ExecuteNonQueryAsync(command);

return rowAffected;

}

}

public async Task<int> UpdateTable(string storedProcName, params SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcName))

{

if (parameters != null)

{

command.Parameters.AddRange(parameters);

}

int rowAffected = await ExecuteNonQueryAsync(command);

return rowAffected;

}

}

// ExecuteScalar methods:

//public async Task<object> ExecuteScalarAsync(SqlCommand command)

//{

// await EnsureConnection();

// return await command.ExecuteScalarAsync();

//}

public async Task<String> ExecuteScalarString(SqlCommand command)

{

await EnsureConnection();

return (string)await command.ExecuteScalarAsync();

}

public async Task<int> ExecuteScalarInt(SqlCommand command)

{

await EnsureConnection();

return (int)await command.ExecuteScalarAsync();

}

public async Task<int> ExecuteIntScalar(string storeProcName, params SqlParameter[] parameters)

{

await EnsureConnection();

using(var cmd = CreateStoredProcedureCommand(storeProcName))

{

if(parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

int variable = (int)await ExecuteScalarInt(cmd);

return variable;

}

}

public async Task<string> ExecuteStringScalar(string storeProcName, params SqlParameter[] parameters)

{

await EnsureConnection();

using (var cmd = CreateStoredProcedureCommand(storeProcName))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

string variable = (string)await ExecuteScalarString(cmd);

return variable;

}

}

// IDataReader Methods:

// use following methods:

public async Task<IDataReader> ExecuteIDataReader(SqlCommand command)

{

await EnsureConnection();

return await command.ExecuteReaderAsync(CommandBehavior.CloseConnection);

}

public async Task<IDataReader> IDataReaderAsync(string storeProcName, params SqlParameter[] parameters )

{

await EnsureConnection();

using(var cmd = CreateStoredProcedureCommand(storeProcName))

{

if (parameters !=null)

{

cmd.Parameters.AddRange(parameters);

}

return await ExecuteIDataReader(cmd);

}

}

public async Task<DataRowCollection> IDataReaderMultiRow(string storeProcName, params SqlParameter[] parameters)

{

using(IDataReader reader = await IDataReaderAsync(storeProcName, parameters))

{

var dataTable = new DataTable();

dataTable.Load(reader);

return dataTable.Rows;

}

}

public async Task<List<DataRow>> IDataReaderMultiRows(string storeProcName, params SqlParameter[] parameters)

{

using (IDataReader reader = await IDataReaderAsync(storeProcName, parameters))

{

var dataTable = new DataTable();

dataTable.Load(reader);

return dataTable.Rows.Cast<DataRow>().ToList();

}

}

public async Task<List<IDataRecord>> IDataRecordsData(string storedProcName, params SqlParameter[] parameters)

{

var records = new List<IDataRecord>();

await EnsureConnection();

using (var cmd = CreateStoredProcedureCommand(storedProcName))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

using (IDataReader reader = await ExecuteIDataReader(cmd))

{

while (reader.Read())

{

records.Add(reader);

}

}

}

return records;

}

// DAPPER :

// Dynamic paramter : INSERT , UPDATE , DELETE

public async Task<int> DapperExecuteNonQueryAsync(string storedProcedure, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

int rowsAffected = await connection.ExecuteAsync(storedProcedure, parameters, commandType: CommandType.StoredProcedure);

return rowsAffected;

}

}

// USING: Get details by Id, or other values: Single row value

//

public async Task<T> DapperQuerySingleRow<T>(string storedProcName, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

var resultData = await \_connection.QuerySingleOrDefaultAsync<T>(storedProcName, parameters, commandType: CommandType.StoredProcedure);

return resultData;

}

}

// USING : Get multiple rows data

//public async Task<IEnumerable<T>> DapperListTableValues<T>(string storedProcName, DynamicParameters parameters)

public async Task<List<T>> DapperMultipleRows<T>(string storedProcName, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

var resultData = await \_connection.QueryAsync<T>(storedProcName,parameters,commandType: CommandType.StoredProcedure);

return resultData.ToList();

}

}

// USING : Get Single Value that may be int or string.

public async Task<T> DapperGetSingleValue<T>(string storedProcName, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

var resutl = await \_connection.QueryFirstOrDefaultAsync<T>(storedProcName, parameters, commandType: CommandType.StoredProcedure);

return resutl;

}

}

// using dataset:

//public async Task<DataSet> DapperMultiDataTableResult(string[] storedProcNames, DynamicParameters parameters)

//{

// var dataset = new DataSet();

// using (var connection = \_connection)

// {

// await EnsureConnection();

// var tasks = storedProcNames.Select(async (procName) =>

// {

// using (var multisetResult = await \_connection.QueryMultipleAsync(procName, parameters, commandType: CommandType.StoredProcedure))

// {

// var dataTable = new DataTable();

// var result = (await Task.Run(() => multisetResult.Read().ToList())).Cast<object>().ToList();

// if (result.Any())

// {

// var properties = result.GetType().GetProperties();

// dataTable.Columns.AddRange(properties.Select(prop => new DataColumn(prop.Name, prop.PropertyType)).ToArray());

// var rows = result.Select(row => properties.Select(prop => prop.GetValue(row)).ToArray()).ToArray();

// rows.Select(row => dataTable.Rows.Add(row)).ToList();

// dataset.Tables.Add(dataTable);

// }

// }

// });

// await Task.WhenAll(tasks);

// }

// return dataset;

//}

}

}

Usage of SqlHelpers methods in following repository:

using Dapper;

using Microsoft.AspNetCore.Mvc.TagHelpers.Cache;

using System.Data;

using System.Data.SqlClient;

using System.Runtime.InteropServices;

namespace JWTRoleAuthentication.CommonLayer.Models

{

//public class SqlHelpers : IDisposable

public class SqlHelpers : IAsyncDisposable

{

private readonly string \_connectionString;

private SqlConnection \_connection;

public SqlHelpers(string connectionString)

{

\_connectionString = connectionString;

\_connection = new SqlConnection(connectionString);

}

//public void Dispose()

//{

// if (\_connection != null)

// {

// if(\_connection.State != System.Data.ConnectionState.Closed)

// {

// \_connection.Close();

// }

// \_connection.Dispose();

// \_connection = null;

// }

//}

public async ValueTask DisposeAsync()

{

if (\_connection != null)

{

if (\_connection.State != System.Data.ConnectionState.Closed)

{

await \_connection.CloseAsync();

}

\_connection.Dispose();

\_connection = null;

}

}

public async Task EnsureConnection()

{

if (\_connection != null)

{

if(\_connection.State != ConnectionState.Open)

{

await \_connection.OpenAsync();

}

}

}

// sql command:

public SqlCommand CreateCommand(string commandText, CommandType commandType = CommandType.StoredProcedure)

{

var command = \_connection.CreateCommand();

command.CommandText = commandText;

command.CommandType = commandType;

return command;

}

// Create StoredProcedure command:

//public SqlCommand CreateStoredProcedureCommand(string storedProcedureName)

//{

// return CreateCommand(storedProcedureName,CommandType.StoredProcedure);

//}

public SqlCommand CreateStoredProcedureCommand(string storedProcedureName)

{

//var command = \_connection.CreateCommand();

//command.CommandText = storedProcedureName;

//command.CommandType = CommandType.StoredProcedure;

//return command;

var command = new SqlCommand(storedProcedureName, \_connection);

command.CommandType = CommandType.StoredProcedure;

return command;

}

// SqlDataAdapter and SqlDataReader

// To Read Table Data using SqlDataAdapter and SqlDataReader for

public async Task<DataTable> ExecuteDataTableAsync(SqlCommand command)

{

DataTable dataTable = new DataTable();

await EnsureConnection();

using(var adapter = new SqlDataAdapter(command))

{

await Task.Run(() => adapter.Fill(dataTable));

return dataTable;

}

}

public async Task<DataTable> ReadDataTable(SqlCommand command)

{

DataTable dataTable = new DataTable();

await EnsureConnection();

using(var reader = await ExecuteReader(command))

{

dataTable.Load(reader);

return dataTable;

}

}

DataSet:

public async Task<DataSet> GetMultipleTablesData(string[] storedProcNames, SqlParameter[] parameters = null)

{

await EnsureConnection();

var dataSet = new DataSet();

var tasks = storedProcNames.Select(async procName =>

{

using (var cmd = CreateStoredProcedureCommand(procName))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

using (var adapter = new SqlDataAdapter(cmd))

{

var dataTable = new DataTable();

await Task.Run(() => adapter.Fill(dataTable));

dataSet.Tables.Add(dataTable);

}

}

});

await Task.WhenAll(tasks);

return dataSet;

}

// To Read database records using SqlDataReader: for read single row and multiple rows and datarowcollections.

// use following methods:

public async Task<SqlDataReader> ExecuteReader(SqlCommand command)

{

// await EnsureConnection();

//return await command.ExecuteReaderAsync(CommandBehavior.CloseConnection);

return await command.ExecuteReaderAsync();

}

public async Task<DataRow> GetSingleRow(string storedProcedureName, params SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcedureName))

{

command.Parameters.AddRange(parameters);

using (var reader = await ExecuteReader(command))

{

var dataTable = new DataTable();

dataTable.Load(reader);

if (dataTable.Rows.Count > 0)

{

return dataTable.Rows[0];

}

}

}

return null;

}

public async Task<List<DataRow>> GetMultipleRows(string storedProcedureName, SqlParameter[] parameters)

{

await EnsureConnection();

var result = new List<DataRow>();

using (var command = CreateStoredProcedureCommand(storedProcedureName))

{

if(parameters != null)

{

command.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(command))

{

var dataTable = new DataTable();

dataTable.Load(reader);

if(dataTable.Rows.Count > 0)

{

result = dataTable.AsEnumerable().Select((row) => row).ToList();

}

}

}

return result;

}

public async Task<DataRowCollection> GetMultipleDataRows(string storedProcedureName, SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcedureName))

{

if (parameters != null)

{

command.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(command))

{

var dataTable = new DataTable();

dataTable.Load(reader);

if(dataTable.Rows.Count > 0)

{

return dataTable.Rows;

}

}

}

return null;

}

// ExecuteNonQuery: for Update and Insert Data Records

// using following methods:

public async Task<int> ExecuteNonQueryAsync(SqlCommand command)

{

await EnsureConnection();

return await command.ExecuteNonQueryAsync();

}

public async Task<int> InsertTable(string storedProcName,params SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcName))

{

if(parameters != null)

{

command.Parameters.AddRange(parameters);

}

int rowAffected = await ExecuteNonQueryAsync(command);

return rowAffected;

}

}

public async Task<int> UpdateTable(string storedProcName, params SqlParameter[] parameters)

{

await EnsureConnection();

using (var command = CreateStoredProcedureCommand(storedProcName))

{

if (parameters != null)

{

command.Parameters.AddRange(parameters);

}

int rowAffected = await ExecuteNonQueryAsync(command);

return rowAffected;

}

}

// ExecuteScalar methods:

//public async Task<object> ExecuteScalarAsync(SqlCommand command)

//{

// await EnsureConnection();

// return await command.ExecuteScalarAsync();

//}

public async Task<String> ExecuteScalarString(SqlCommand command)

{

await EnsureConnection();

return (string)await command.ExecuteScalarAsync();

}

public async Task<int> ExecuteScalarInt(SqlCommand command)

{

await EnsureConnection();

return (int)await command.ExecuteScalarAsync();

}

public async Task<int> ExecuteIntScalar(string storeProcName, params SqlParameter[] parameters)

{

await EnsureConnection();

using(var cmd = CreateStoredProcedureCommand(storeProcName))

{

if(parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

int variable = (int)await ExecuteScalarInt(cmd);

return variable;

}

}

public async Task<string> ExecuteStringScalar(string storeProcName, params SqlParameter[] parameters)

{

await EnsureConnection();

using (var cmd = CreateStoredProcedureCommand(storeProcName))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

string variable = (string)await ExecuteScalarString(cmd);

return variable;

}

}

// IDataReader Methods:

// use following methods:

public async Task<IDataReader> ExecuteIDataReader(SqlCommand command)

{

await EnsureConnection();

return await command.ExecuteReaderAsync(CommandBehavior.CloseConnection);

}

public async Task<IDataReader> IDataReaderAsync(string storeProcName, params SqlParameter[] parameters )

{

await EnsureConnection();

using(var cmd = CreateStoredProcedureCommand(storeProcName))

{

if (parameters !=null)

{

cmd.Parameters.AddRange(parameters);

}

return await ExecuteIDataReader(cmd);

}

}

public async Task<DataRowCollection> IDataReaderMultiRow(string storeProcName, params SqlParameter[] parameters)

{

using(IDataReader reader = await IDataReaderAsync(storeProcName, parameters))

{

var dataTable = new DataTable();

dataTable.Load(reader);

return dataTable.Rows;

}

}

public async Task<List<DataRow>> IDataReaderMultiRows(string storeProcName, params SqlParameter[] parameters)

{

using (IDataReader reader = await IDataReaderAsync(storeProcName, parameters))

{

var dataTable = new DataTable();

dataTable.Load(reader);

return dataTable.Rows.Cast<DataRow>().ToList();

}

}

public async Task<List<IDataRecord>> IDataRecordsData(string storedProcName, params SqlParameter[] parameters)

{

var records = new List<IDataRecord>();

await EnsureConnection();

using (var cmd = CreateStoredProcedureCommand(storedProcName))

{

if (parameters != null)

{

cmd.Parameters.AddRange(parameters);

}

using (IDataReader reader = await ExecuteIDataReader(cmd))

{

while (reader.Read())

{

records.Add(reader);

}

}

}

return records;

}

// DAPPER :

// Dynamic paramter : INSERT , UPDATE , DELETE

public async Task<int> DapperExecuteNonQueryAsync(string storedProcedure, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

int rowsAffected = await connection.ExecuteAsync(storedProcedure, parameters, commandType: CommandType.StoredProcedure);

return rowsAffected;

}

}

// USING: Get details by Id, or other values: Single row value

//

public async Task<T> DapperQuerySingleRow<T>(string storedProcName, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

var resultData = await \_connection.QuerySingleOrDefaultAsync<T>(storedProcName, parameters, commandType: CommandType.StoredProcedure);

return resultData;

}

}

// USING : Get multiple rows data

//public async Task<IEnumerable<T>> DapperListTableValues<T>(string storedProcName, DynamicParameters parameters)

public async Task<List<T>> DapperMultipleRows<T>(string storedProcName, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

var resultData = await \_connection.QueryAsync<T>(storedProcName,parameters,commandType: CommandType.StoredProcedure);

return resultData.ToList();

}

}

// USING : Get Single Value that may be int or string.

public async Task<T> DapperGetSingleValue<T>(string storedProcName, DynamicParameters parameters)

{

using (var connection = \_connection)

{

await EnsureConnection();

var resutl = await \_connection.QueryFirstOrDefaultAsync<T>(storedProcName, parameters, commandType: CommandType.StoredProcedure);

return resutl;

}

}

}

}

Usage of SqlHelpers method in following repository :

CustomerDal.cs:

using JWTRoleAuthentication.CommonLayer.Models;

using Microsoft.AspNetCore.Mvc.TagHelpers.Cache;

using Microsoft.Extensions.Options;

using System.Data;

using System.Data.SqlClient;

using System.Runtime.InteropServices;

namespace JWTRoleAuthentication.JWTDAL

{

public class CustomerDAL

{

private readonly IOptions<ConnectionStrings> \_options;

private readonly SqlHelpers \_sqlHelpers;

public CustomerDAL(IOptions<ConnectionStrings> options)

{

\_options = options;

// passing connection string value for SqlHelpers method.

\_sqlHelpers = new SqlHelpers(\_options.Value.AuthDBCon.ToString());

}

//public async Task<DataRow> GetCustomerDetailsById(int id)

//{

// var storedProcName = "GetCustomerById";

// return await \_sqlHelpers.GetSingleRow(storedProcName, new SqlParameter("@Id", id));

//}

//public async Task<DataSet> GetMultipleTablesData()

//{

// string[] storedProcNames =

// {

// "GetUserDetails",

// "GetCustomers"

// };

// return await \_sqlHelpers.GetMultipleTablesData(storedProcNames, null);

//}

// create mapcustomer model for retrive the data using IDataRecord:

private Task<CustomerModel> MapCustomer(IDataRecord record)

{

var model = new CustomerModel

{

Id = Convert.ToInt32(record["Id"]),

UserName = record["UserName"].ToString(),

Email = record["Email"].ToString(),

MobileNumber = record["MobileNumber"].ToString(),

Address = record["Address"].ToString()

};

Console.WriteLine(model);

return Task.FromResult(model);

}

// create map customers list using IDataRecord

private async Task<List<CustomerModel>> MapCustomerList(IDataRecord record)

{

var customerList = new List<CustomerModel>();

customerList.Add(new CustomerModel

{

Id = Convert.ToInt32(record["Id"]),

UserName = record["UserName"].ToString(),

Email = record["Email"].ToString(),

MobileNumber = record["MobileNumber"].ToString(),

Address = record["Address"].ToString()

});

return await Task.FromResult(customerList);

}

public async Task<CustomerModel> GetCustomerDetailsById(int id)

{

try

{

// retrive the single row data using stored procedure name and parameters.

var storedProcName = "GetCustomerById";

var result = await \_sqlHelpers.GetSingleRow(storedProcName, new SqlParameter("@Id", id));

var customer = new CustomerModel

{

Id = Convert.ToInt32(result["Id"]),

UserName = result["UserName"].ToString(),

Email = result["Email"].ToString(),

MobileNumber = result["MobileNumber"].ToString(),

Address = result["Address"].ToString()

};

// iDataReader:

var iDataReader = await \_sqlHelpers.IDataReaderAsync(storedProcName, new SqlParameter("@Id", id));

if(iDataReader.Read())

{

var model = new CustomerModel

{

Id = Convert.ToInt32(iDataReader["Id"]),

UserName = iDataReader["UserName"].ToString(),

Email = iDataReader["Email"].ToString(),

MobileNumber = iDataReader["MobileNumber"].ToString(),

Address = iDataReader["Address"].ToString()

};

Console.WriteLine(model);

// IDataRecord:

await MapCustomer((IDataRecord)iDataReader);

}

// using dapper:

var parameters = new DynamicParameters();

parameters.Add("@Id", id, DbType.Int32);

var getCustomerDetails = await \_sqlHelpers.DapperQuerySingleRow<CustomerModel>(storedProcName, parameters);

// return getCustomerDetails;

return customer;

}

catch (Exception ex)

{

throw ex;

}

}

public async Task<DatasetResponse> GetMultipleTablesData()

{

DatasetResponse datasetResponse = null;

datasetResponse = new DatasetResponse();

string[] storedProcNames =

{

"GetUserDetails",

"GetCustomers"

};

// Dataset :

var dataset = await \_sqlHelpers.GetMultipleTablesData(storedProcNames, null);

try

{

var userTable = dataset.Tables[0];

var customerTable = dataset.Tables[1];

datasetResponse.users = (from DataRow r in userTable.Rows

select new Register

{

UserID = (Guid)r["UserID"],

FirstName = r["FirstName"].ToString(),

LastName = r["LastName"].ToString(),

UserName = r["UserName"].ToString(),

Email = r["Email"].ToString(),

DateOfBirth = Convert.ToDateTime(r["DateOfBirth"]),

MobileNumber = r["MobileNumber"].ToString(),

ZipCode = r["ZipCode"].ToString(),

StoreID = Convert.ToInt32(r["StoreID"]),

Role = r["Role"].ToString()

}

).ToList();

datasetResponse.Customer = (from DataRow dt in customerTable.Rows select new CustomerModel

{

Id = Convert.ToInt32(dt["Id"]),

UserName = dt["UserName"].ToString(),

Email = dt["Email"].ToString(),

MobileNumber = dt["MobileNumber"].ToString(),

Address = dt["Address"].ToString()

}).ToList();

datasetResponse.StatusCode = 200;

datasetResponse.StatusMessage= "OK";

}catch (Exception ex)

{

datasetResponse.StatusCode = 500;

datasetResponse.StatusMessage = "Something went wrong.";

}

return datasetResponse;

}

public async Task<string> GetCustomerEmail(int id)

{

var storedProcName = "GetCustomerEmail";

// ExecuteScalar with string value:

using (var cmd = \_sqlHelpers.CreateStoredProcedureCommand(storedProcName))

{

cmd.Parameters.AddWithValue("@Id", id);

string email = await \_sqlHelpers.ExecuteScalarString(cmd);

return email;

}

// USING DAPPER:

var parameter = new DynamicParameters();

parameter.Add("@Id",id,DbType.Int32);

var getEmail = await \_sqlHelpers.DapperGetSingleValue<string>(storedProcName,parameter);

return getEmail;

}

public async Task<List<CustomerModel>> GetCustomers()

{

try

{

var customers = new List<CustomerModel>();

var listCustomers = new List<CustomerModel>();

var storedProcName = "GetCustomers";

// using multiple Rows:

// GetMultipleRows with SqlDataReader

var customerTable = await \_sqlHelpers.GetMultipleRows(storedProcName, null);

// GetMultipleDataRows with SqlDataAdapter

var customerTable = await \_sqlHelpers.GetMultipleDataRows(storedProcName, null);

customers = (from DataRow dr in customerTable

select new CustomerModel

{

Id = Convert.ToInt32(dr["Id"]),

UserName = dr["UserName"].ToString(),

Email = dr["Email"].ToString(),

MobileNumber = dr["MobileNumber"].ToString(),

Address = dr["Address"].ToString()

}).ToList();

// IDataReader for multiple data rows:

//var iDataReader = await \_sqlHelpers.IDataReaderAsync(storedProcName, null);

//while (iDataReader.Read())

//{

// listCustomers.Add(new CustomerModel

// {

// Id = Convert.ToInt32(iDataReader["Id"]),

// UserName = iDataReader["UserName"].ToString(),

// Email = iDataReader["Email"].ToString(),

// MobileNumber = iDataReader["MobileNumber"].ToString(),

// Address = iDataReader["Address"].ToString()

// });

// //await MapCustomerList((IDataRecord)iDataReader);

//}

// var iDataReader = await \_sqlHelpers.IDataReaderMultiRows(storedProcName, null);

//listCustomers = (from DataRow dr in iDataReader

// select new CustomerModel

// {

// Id = Convert.ToInt32(dr["Id"]),

// UserName = dr["UserName"].ToString(),

// Email = dr["Email"].ToString(),

// MobileNumber = dr["MobileNumber"].ToString(),

// Address = dr["Address"].ToString()

// }

// ).ToList();

// var IDataRecordsData = await \_sqlHelpers.IDataRecordsData(storedProcName, null);

// IDataRecord for list of customer

// var records = IDataRecordsData.Select(async record => await MapCustomerList(record)).ToList();

//using (var cmd = \_sqlHelpers.CreateStoredProcedureCommand(storedProcName))

//{

// using adapter:

// var customerTable = await \_sqlHelpers.ExecuteDataTableAsync(cmd);

// using reader:

// var customerTable = await \_sqlHelpers.ReadDataTable(cmd);

// customers = (from DataRow dr in customerTable.Rows

// select new CustomerModel

// {

// Id = Convert.ToInt32(dr["Id"]),

// UserName = dr["UserName"].ToString(),

// Email = dr["Email"].ToString(),

// MobileNumber = dr["MobileNumber"].ToString(),

// Address = dr["Address"].ToString()

// }).ToList();

//}

// using dapper get customer list data:

var customersList = await \_sqlHelpers.DapperMultipleRows<CustomerModel>(storedProcName,null);

return customersList;

//return customers;

}

catch (Exception ex)

{

throw ex;

}

}

public async Task<CustomerResponse> UpdateCustomer(int id, CustomerModel model)

{

CustomerResponse customerResponse = new CustomerResponse();

try

{

var storedProcName = "UpdateCustomer";

// using sqlparameter:

//var parameters = new SqlParameter[]

//{

// new SqlParameter ("@Id",id),

// new SqlParameter("@UserName",model.UserName),

// new SqlParameter("@Email",model.Email),

// new SqlParameter("@MobileNumber",model.MobileNumber),

// new SqlParameter("@Address",model.Address),

//};

// Update records

// int rowsAffected = await \_sqlHelpers.UpdateTable(storedProcName, parameters);

// Update records using Dapper:

var parameters = new DynamicParameters();

parameters.Add("@Id", id, DbType.Int32);

parameters.Add("@UserName", model.UserName, DbType.String, size: 100);

parameters.Add("@Email", model.Email, DbType.String, size: 100);

parameters.Add("@MobileNumber", model.MobileNumber, DbType.String, size: 100);

parameters.Add("@Address", model.Address, DbType.String, size: 100);

//var rowsAffected = await \_sqlHelpers.DapperExecuteNonQueryAsync(storedProcName, parameters);

int rowsAffected = await \_sqlHelpers.DapperExecuteNonQueryAsync(storedProcName,parameters);

if (rowsAffected > 0)

{

customerResponse.Customer = new CustomerModel();

customerResponse.Customer.Id = id;

customerResponse.Customer.UserName = model.UserName;

customerResponse.Customer.Email = model.Email;

customerResponse.Customer.MobileNumber = model.MobileNumber;

customerResponse.Customer.Address = model.Address;

customerResponse.StatusCode = 200;

customerResponse.StatusMessage = "Customer has been updated successfully.";

}

}

catch (Exception ex)

{

customerResponse.StatusCode = 500;

customerResponse.StatusMessage = "Something went wrong.";

}

return customerResponse;

}

public async Task<CustomerResponse> AddCustomer(CustomerModel model)

{

CustomerResponse customerResponse = new CustomerResponse();

try

{

var storedProcName = "InsertCustomer";

// USING SQL COMMANDS:

// var parameters = new SqlParameter[]

// {

// new SqlParameter("@UserName",model.UserName),

// new SqlParameter("@Email",model.Email),

// new SqlParameter("@MobileNumber",model.MobileNumber),

// new SqlParameter("@Address",model.Address),

// };

// int rowsAffected = await \_sqlHelpers.InsertTable(storedProcName, parameters);

// USING DAPPER:

var parameters = new DynamicParameters();

parameters.Add("@UserName", model.UserName, DbType.String, size:100);

parameters.Add("@Email", model.Email, DbType.String, size: 100);

parameters.Add("@MobileNumber", model.MobileNumber, DbType.String, size: 100);

parameters.Add("@Address", model.Address, DbType.String, size: 100);

var affectedRow = await \_sqlHelpers.DapperExecuteNonQueryAsync(storedProcName,parameters);

if(affectedRow > 0)

{

customerResponse.Customer = model;

customerResponse.StatusCode = 200;

customerResponse.StatusMessage = "Customer details has been add successfully.";

}

else

{

// customerResponse.Customer = null;

customerResponse.StatusCode = 400;

customerResponse.StatusMessage = "Insert of customer details has been failed.";

}

}catch(Exception ex)

{

customerResponse.StatusCode = 500;

customerResponse.StatusMessage = "Something went wrong. Please try again later.";

}

return customerResponse;

}