RSASupportAPI

Controller:

using Microsoft.AspNetCore.Http;

using Microsoft.AspNetCore.Mvc;

using RSASupportAPI.Models;

using RSASupportAPI.RSASupportBLL;

namespace RSASupportAPI.Controllers

{

[Route("api/[controller]")]

[ApiController]

public class RSASupportAPIController : ControllerBase

{

private readonly IClientService \_clientService;

public RSASupportAPIController(IClientService clientService)

{

\_clientService = clientService;

}

[HttpGet]

[Route("GetClients")]

public async Task<ClinetResponse> GetClients()

{

ClinetResponse clinetResponse = new ClinetResponse();

var result = await \_clientService.GetClients();

try

{

if (result == null) {

clinetResponse.ErroCode = "404";

clinetResponse.ErrorMessage = "Clients not found.";

return clinetResponse;

}

clinetResponse.ErroCode = "200";

clinetResponse.ErrorMessage = "Success";

clinetResponse.Clients = result;

}

catch (Exception ex) {

clinetResponse.ErroCode = "500";

clinetResponse.ErrorMessage = "Something went wrong. Please try again.";

}

return clinetResponse;

}

[HttpGet]

[Route("GetClientData")]

public async Task<IActionResult> GetClientData(string clientName, string selectQuery)

{

try

{

var result = await \_clientService.GetClientData(clientName, selectQuery);

if (result == null)

{

return new JsonResult( new

{

ErrorCode = StatusCodes.Status404NotFound,

ErrorMessage = "Client data not found."

});

}

return new JsonResult(new

{

ErrorCode = StatusCodes.Status200OK,

ErrorMessage = "Successful",

ClientInfo = result

});

}

catch (Exception ex)

{

return StatusCode(StatusCodes.Status500InternalServerError, ex.Message);

}

}

//public async Task<ClientDataResponse> GetClientData(string clientName, string selectQuery)

//{

// ClientDataResponse clientDataResponse = new ClientDataResponse();

// try

// {

// var result = await \_clientService.GetClientData(clientName, selectQuery);

// if (result == null) {

// clientDataResponse.ErroCode = "404";

// clientDataResponse.ErrorMessage = "ClientData not found.";

// return clientDataResponse;

// }

// clientDataResponse.ErroCode = "200";

// clientDataResponse.ErrorMessage = "Success";

// clientDataResponse.ClientInfo = result;

// }

// catch (Exception ex)

// {

// clientDataResponse.ErroCode = "500";

// clientDataResponse.ErrorMessage = "Something went wrong. Please try again.";

// }

// return clientDataResponse;

//}

}

}

Helpers

using System.Security.Cryptography;

using System.Text;

namespace RSASupportAPI.Models

{

public class Helpers

{

public Helpers()

{

}

public async Task<string> ConnectionStringBuilder(RSADBConnection model)

{

string connectionString = "";

try

{

connectionString = "Data Source =" + model.DataSource + ";Initial Catalog=" + model.Database + ";uid=" + model.UserId + ";Password=" + model.Password;

}

catch (Exception ex) {

connectionString = "";

}

return await Task.FromResult(connectionString);

}

// Encrypt: AES 256

public async Task<string> EncryptAes(string plainText)

{

byte[] key = Encoding.UTF8.GetBytes("0123456789abcdef0123456789abcdef"); // 32-byte key

byte[] iv = Encoding.UTF8.GetBytes("abcdef9876543210"); // 16-byte IV

using (Aes aesAlg = Aes.Create())

{

aesAlg.Key = key;

aesAlg.IV = iv;

aesAlg.Padding = PaddingMode.PKCS7;

ICryptoTransform encryptor = aesAlg.CreateEncryptor(aesAlg.Key, aesAlg.IV);

using (var msEncrypt = new MemoryStream())

{

using (var csEncrypt = new CryptoStream(msEncrypt, encryptor, CryptoStreamMode.Write))

{

using (var swEncrypt = new StreamWriter(csEncrypt))

{

await swEncrypt.WriteAsync(plainText);

}

}

byte[] encrypted = msEncrypt.ToArray();

return Convert.ToBase64String(encrypted); // Return Base64 encoded string

}

}

}

// Decrypt AES: 256

public async Task<string> DecryptAes(string base64CipherText)

{

byte[] key = Encoding.UTF8.GetBytes("0123456789abcdef0123456789abcdef"); // 32-byte key

byte[] iv = Encoding.UTF8.GetBytes("abcdef9876543210"); // 16-byte IV

try

{

byte[] cipherTextBytes = Convert.FromBase64String(base64CipherText);

using (Aes aesAlg = Aes.Create())

{

aesAlg.Key = key;

aesAlg.IV = iv;

aesAlg.Padding = PaddingMode.PKCS7;

ICryptoTransform decryptor = aesAlg.CreateDecryptor(aesAlg.Key, aesAlg.IV);

using (var msDecrypt = new MemoryStream(cipherTextBytes))

{

using (var csDecrypt = new CryptoStream(msDecrypt, decryptor, CryptoStreamMode.Read))

{

using (var msPlain = new MemoryStream())

{

await csDecrypt.CopyToAsync(msPlain);

byte[] decryptedBytes = msPlain.ToArray();

return Encoding.UTF8.GetString(decryptedBytes); // Return decrypted string

}

}

}

}

}

catch (CryptographicException cryptoEx)

{

throw new Exception("Decryption failed: " + cryptoEx.Message);

}

}

// Encrypt password: Base64

public string EncryptPassword(string password)

{

if (string.IsNullOrEmpty(password))

{

return "";

}

else

{

byte[] base64Password = ASCIIEncoding.ASCII.GetBytes(password);

string encryptedPassword = Convert.ToBase64String(base64Password);

return encryptedPassword;

}

}

// Decrypt Password: Base64

public string DecryptPassword(string password) {

if (string.IsNullOrEmpty(password)) {

return "";

}

else

{

byte[] encryptedPassword = Convert.FromBase64String(password);

string decryptedPassword = ASCIIEncoding.ASCII.GetString(encryptedPassword);

return decryptedPassword;

}

}

}

}

IClientRepo.cs

using RSASupportAPI.Models;

namespace RSASupportAPI.RSASupportDAL

{

public interface IClientRepo

{

Task<Client> GetClients();

Task<List<dynamic>> GetClientData(string clientName, string selectQuery);

//Task<ClinetResponse> GetClients();

//Task<UserDetails> GetClientData(string clientName,string selectQuery);

// Task<Dictionary<string, object>> GetClientData(string clientName, string selectQuery);

//Task<dynamic> GetClientData(string clientName, string selectQuery);

}

}

ClientRepo:

using Amazon.S3;

using Newtonsoft.Json;

using RSASupportAPI.Models;

using System.Data.SqlClient;

using System.Security.Cryptography;

namespace RSASupportAPI.RSASupportDAL

{

public class ClientRepo : IClientRepo

{

private readonly List<AppConfigurations> \_appConfigurations;

private readonly Helpers \_helpers;

private readonly IAmazonS3 \_amazonS3;

private readonly S3BucketHelpers \_s3BucketHelpers;

public ClientRepo(

List<AppConfigurations> appConfigurations,

Helpers helpers,

IAmazonS3 amazonS3,

S3BucketHelpers s3BucketHelpers

)

{

\_appConfigurations = appConfigurations;

\_helpers = helpers;

\_amazonS3 = amazonS3;

\_s3BucketHelpers = s3BucketHelpers;

}

public async Task<Client> GetClients()

{

var client = new Client();

//var bucketName = "rsasupportapi";

//var s3FolderFilePath = "RSASupportConfigurations/AppConfigurations.json";

//var validatePath = await \_s3BucketHelpers.ExistedFolderPathKey(bucketName, s3FolderFilePath);

//if (validatePath != null) {

// var getJsonData = await \_s3BucketHelpers.S3JsonConfigData(bucketName, s3FolderFilePath);

// //var getJsonData = \_appConfigurations.ToList();

// var getClientNames = getJsonData.Select(c => c.ClinetName

// ).ToList();

// client.clientNames = getClientNames;

//}

var getJsonData = await \_s3BucketHelpers.GetS3JsonFileData();

var getClientNames = getJsonData.Select(c => c.ClinetName

).ToList();

client.clientNames = getClientNames;

return await Task.FromResult(client);

}

public async Task<List<dynamic>> GetClientData(string clientName, string selectQuery)

{

var appconfigurations = new List<AppConfigurations>();

appconfigurations = await \_s3BucketHelpers.GetS3JsonFileData();

if (appconfigurations != null && appconfigurations.Count > 0)

{

var s3bucketData = appconfigurations?.FirstOrDefault(c => c.ClinetName == clientName);

if (s3bucketData != null)

{

//string plaintext = "sai123";

//string encryptText = await \_helpers.EncryptAes(plaintext);

//string textEncrypt = "JOzvfBDWRbcBMDk9QBDiyg==";

//string decrypt = await \_helpers.DecryptAes(encryptText);

string decryptText = await \_helpers.DecryptAes(s3bucketData.Password);

var rsaDBCon = new RSADBConnection

{

DataSource = s3bucketData.DBInstanceName,

Database = s3bucketData.DBName,

UserId = s3bucketData.UserName,

// Password =\_helpers.DecryptPassword(s3bucketData.Password),

Password = decryptText

};

string connectionString = await \_helpers.ConnectionStringBuilder(rsaDBCon);

SqlHelpers sqlHelpers = new SqlHelpers(connectionString);

//var row = await sqlHelpers.GetDynamicDataAsJson(selectQuery,null);

dynamic result = await sqlHelpers.GetDynamicQueryTableDataWithMultipleRows(selectQuery, null);

return result;

}

}

return null;

}

//public async Task<UserDetails> GetClientData(string clientName, string selectQuery)

//{

// //var bucketName = "rsasupportapi";

// //var s3FolderFilePath = "RSASupportConfigurations/AppConfigurations.json";

// //var validatePath = await \_s3BucketHelpers.ExistedFolderPathKey(bucketName, s3FolderFilePath);

// //var appconfigurations = new List<AppConfigurations>();

// //if (validatePath != null) {

// // //var jsonData = await \_s3BucketHelpers.GetS3BucketObjectDetails(bucketName, s3FolderFilePath);

// // //appconfigurations = JsonConvert.DeserializeObject<List<AppConfigurations>>(jsonData);

// // appconfigurations = await \_s3BucketHelpers.S3JsonConfigData(bucketName, validatePath);

// //}

// UserDetails userDetails = null;

// userDetails = new UserDetails();

// var appconfigurations = new List<AppConfigurations>();

// appconfigurations = await \_s3BucketHelpers.GetS3JsonFileData();

// if (appconfigurations != null && appconfigurations.Count > 0) {

// var s3bucketData = appconfigurations?.FirstOrDefault(c => c.ClinetName == clientName);

// if (s3bucketData != null) {

// var rsaDBCon = new RSADBConnection

// {

// DataSource = s3bucketData.DBInstanceName,

// Database = s3bucketData.DBName,

// UserId = s3bucketData.UserName,

// Password = s3bucketData.Password,

// };

// string connectionString = await \_helpers.ConnectionStringBuilder(rsaDBCon);

// SqlHelpers sqlHelpers = new SqlHelpers(connectionString);

// //var row = await sqlHelpers.GetDynamicDataAsJson(selectQuery,null);

// //var table = await sqlHelpers.GetFirstRowAsDictionary(selectQuery, null);

// dynamic data = await sqlHelpers.GetDynamicQueryTableData(selectQuery, null);

// var dataRows = await sqlHelpers.GetSelectQuerySingleDataRow(selectQuery, null);

// userDetails.ClientId = Convert.ToInt32(dataRows["ClientId"]);

// userDetails.ClientName = dataRows["ClinetName"].ToString() ?? string.Empty;

// userDetails.UserName = dataRows["UserName"].ToString() ?? string.Empty;

// userDetails.FirstName = dataRows["FirstName"].ToString() ?? string.Empty;

// userDetails.LastName = dataRows["LastName"].ToString() ?? string.Empty;

// userDetails.Email = dataRows["Email"].ToString() ?? string.Empty;

// userDetails.MemberNumber = dataRows["MemberNumber"].ToString() ?? string.Empty;

// }

// }

// return userDetails;

//}

//public async Task<Dictionary<string,object >> GetClientData(string clientName, string selectQuery)

// {

// // var table = new Dictionary<string, object>();

// var appconfigurations = new List<AppConfigurations>();

// appconfigurations = await \_s3BucketHelpers.GetS3JsonFileData();

// if (appconfigurations != null && appconfigurations.Count > 0)

// {

// var s3bucketData = appconfigurations?.FirstOrDefault(c => c.ClinetName == clientName);

// if (s3bucketData != null)

// {

// var rsaDBCon = new RSADBConnection

// {

// DataSource = s3bucketData.DBInstanceName,

// Database = s3bucketData.DBName,

// UserId = s3bucketData.UserName,

// Password = s3bucketData.Password,

// };

// string connectionString = await \_helpers.ConnectionStringBuilder(rsaDBCon);

// SqlHelpers sqlHelpers = new SqlHelpers(connectionString);

// //var row = await sqlHelpers.GetDynamicDataAsJson(selectQuery,null);

// // dynamic data = await sqlHelpers.GetDynamicQueryTableData(selectQuery, null);

// // var dataRows = await sqlHelpers.GetSelectQuerySingleDataRow(selectQuery, null);

// var table = await sqlHelpers.GetFirstRowAsDictionary(selectQuery, null);

// return table;

// }

// }

// return null;

// }

}

}

IClientService.cs:

using RSASupportAPI.Models;

namespace RSASupportAPI.RSASupportBLL

{

public interface IClientService

{

Task<Client> GetClients();

Task<List<dynamic>> GetClientData(string clientName, string selectQuery);

// Task<UserDetails> GetClientData(string clientName, string selectQuery);

// Task<Dictionary<string, object>> GetClientData(string clientName, string selectQuery);

// Task<dynamic> GetClientData(string clientName, string selectQuery);

}

}

IClientService

using RSASupportAPI.Models;

using RSASupportAPI.RSASupportDAL;

namespace RSASupportAPI.RSASupportBLL

{

public class ClientService : IClientService

{

private readonly IClientRepo \_clientRepo;

public ClientService(IClientRepo clientRepo)

{

\_clientRepo = clientRepo;

}

// public async Task<Dictionary<string, object>> GetClientData(string clientName, string selectQuery)

// public async Task<dynamic> GetClientData(string clientName, string selectQuery)

public async Task<List<dynamic>> GetClientData(string clientName, string selectQuery)

{

var result = await \_clientRepo.GetClientData(clientName, selectQuery);

return result;

}

//public async Task<UserDetails> GetClientData(string clientName, string selectQuery)

//{

// var result = await \_clientRepo.GetClientData(clientName, selectQuery);

// return result;

//}

public async Task<Client> GetClients()

{

var result = await \_clientRepo.GetClients();

return result;

}

}

}

SQLHelpers:

using System.Data.SqlClient;

using System.Data;

using Newtonsoft.Json;

using System.Dynamic;

namespace RSASupportAPI.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;

}

// create command for select query execution:

public SqlCommand CreateSelectQueryCommand(string query)

{

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

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;

}

// To read record data using select query with single record:

public async Task<DataRow> GetSelectQuerySingleDataRow(string query,SqlParameter[]? parameters = null )

{

await EnsureConnection();

using (var cmd = CreateSelectQueryCommand(query)) {

if(cmd.Parameters.Count > 0)

{

cmd.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(cmd)) {

var dataTable = new DataTable();

dataTable.Load(reader);

if (dataTable.Rows.Count > 0) {

return dataTable.Rows[0];

}

}

}

return null;

}

public async Task<DataTable> GetSelectQuerySingleDataTable(string query, SqlParameter[]? parameters = null)

{

var dataTable = new DataTable();

await EnsureConnection();

using (var cmd = CreateSelectQueryCommand(query))

{

if (cmd.Parameters.Count > 0)

{

cmd.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(cmd))

{

dataTable.Load(reader);

}

}

return dataTable;

}

// single row dynamically:

public async Task<dynamic> GetDynamicQueryTableData(string query, SqlParameter[]? parameters = null)

{

var dataTable =await GetSelectQuerySingleDataTable(query, parameters);

if (dataTable.Rows.Count > 0) {

DataRow row = dataTable.Rows[0];

dynamic result = new ExpandoObject();

var resultDictionary = (IDictionary<string, object>)result;

dataTable.Columns

.Cast<DataColumn>()

.ToList()

.ForEach(column => resultDictionary[column.ColumnName] = row[column]);

return result;

}

return null;

}

// multiple rows dynamically:

public async Task<List<dynamic>> GetDynamicQueryTableDataWithMultipleRows(string query, SqlParameter[]? parameters = null)

{

var dataTable = await GetSelectQuerySingleDataTable(query, parameters);

if(dataTable.Rows.Count > 0)

{

var result = dataTable.AsEnumerable().

Select(

row =>

{

dynamic rowExpand = new ExpandoObject();

var rowDictionary = (IDictionary<string, object>)rowExpand;

dataTable.Columns .Cast<DataColumn>()

.ToDictionary(column => column.ColumnName, column => row[column] )

.ToList()

.ForEach(x => rowDictionary[x.Key] = x.Value);

return rowExpand;

}

)

.ToList();

return result;

}

return null;

}

public async Task<Dictionary<string, object>> GetFirstRowAsDictionary(string query, SqlParameter[]? parameters = null)

{

var dataTable = await GetSelectQuerySingleDataTable(query, parameters);

if (dataTable.Rows.Count > 0)

{

DataRow row = dataTable.Rows[0];

var result = dataTable.Columns.Cast<DataColumn>()

.ToList()

.Select(column => new {column.ColumnName,value= row[column] })

.ToDictionary(x => x.ColumnName,y =>y.value);

return result;

}

return null; // Or return an empty dictionary if preferred

}

public async Task<string> GetDynamicDataAsJson(string query, SqlParameter[]? parameters = null)

{

var results = new List<Dictionary<string, object>>();

await EnsureConnection();

using (var cmd = CreateSelectQueryCommand(query))

{

if (parameters != null && parameters.Length > 0)

{

cmd.Parameters.AddRange(parameters);

}

using (var reader = await ExecuteReader(cmd))

{

if (reader.HasRows)

{

while ( reader.Read())

{

var row = new Dictionary<string, object>();

for (int i = 0; i < reader.FieldCount; i++)

{

string columnName = reader.GetName(i);

object columnValue = reader.GetValue(i);

row[columnName] = columnValue;

}

results.Add(row); // Add the row to the results

}

}

}

}

// Serialize the list of dictionaries to a JSON string

return await Task.FromResult(JsonConvert.SerializeObject(results));

}

public async Task<List<DataRow>> GetSelectQueryMultipleRows(string query, SqlParameter[] parameters)

{

await EnsureConnection();

var result = new List<DataRow>();

using (var command = CreateSelectQueryCommand(query))

{

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<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;

}

}

}

S3BucketHelpers

using Amazon.S3.Model;

using Amazon.S3;

using Newtonsoft.Json;

namespace RSASupportAPI.Models

{

public class S3BucketHelpers

{

private readonly IAmazonS3 \_amazonS3;

public S3BucketHelpers(IAmazonS3 amazonS3)

{

\_amazonS3 = amazonS3;

}

public async Task<string> GetS3BucketObjectDetails(string bucketName, string key)

{

var response = await \_amazonS3.GetObjectAsync(bucketName, key);

using (var reader = new StreamReader(response.ResponseStream))

{

return await reader.ReadToEndAsync();

}

}

public async Task<List<AppConfigurations>> S3JsonConfigData(string bucketName, string key)

{

var configData = new List<AppConfigurations>();

var getS3objects = await GetS3BucketObjectDetails(bucketName,key);

configData = JsonConvert.DeserializeObject<List<AppConfigurations>>(getS3objects);

if (configData != null) {

return configData;

}

return null;

}

public async Task<bool> ValidateKeyPath(string bucketName, string folderName, string transactionDate, int storeId, string fileName)

{

var request = new ListObjectsV2Request

{

BucketName = bucketName,

Prefix = "",

};

var s3BucketObjects = await \_amazonS3.ListObjectsV2Async(request);

var validatePath = s3BucketObjects.S3Objects.FirstOrDefault(x =>

{

return x.Key.Split("/")[0] == folderName &&

x.Key.Split("/")[1].Length > 1 && x.Key.Split("/")[1] == transactionDate &&

x.Key.Split("/")[2].Length > 2 && x.Key.Split("/")[2] == (storeId).ToString() &&

x.Key.Split("/")[3].Length > 3 && x.Key.Split("/")[3] == fileName;

});

if (validatePath != null)

{

return true;

}

return false;

}

public async Task<string> ValidateKeyPathAsync(string bucketName, string folderName, string transactionDate, int storeId, string fileName)

{

string key = string.Empty;

var request = new ListObjectsV2Request

{

BucketName = bucketName,

Prefix = ""

};

var result = await \_amazonS3.ListObjectsV2Async(request);

var validPath = result.S3Objects.FirstOrDefault(x =>

{

return x.Key.Split("/")[0] == folderName &&

x.Key.Split("/")[1] == transactionDate &&

x.Key.Split("/")[2] == storeId.ToString() &&

x.Key.Split("/")[3] == fileName;

});

if (validPath != null)

{

key = validPath.Key;

}

return key;

}

public async Task<bool> ExistFolderPath(string bucketName, string folderPath)

{

var request = new ListObjectsV2Request

{

BucketName = bucketName,

Prefix = folderPath,

};

var result = await \_amazonS3.ListObjectsV2Async(request);

if (result.S3Objects.Count > 0 || result.CommonPrefixes.Count > 0)

{

return true;

}

return false;

}

public async Task<string> ExistedFolderPathKey(string bucketName, string key)

{

string keyPath = string.Empty;

try

{

var request = new ListObjectsV2Request

{

BucketName = bucketName,

Prefix = key,

};

var result = await \_amazonS3.ListObjectsV2Async(request);

if (result.S3Objects.Count > 0)

{

//return keyPath = key;

return keyPath = result.Prefix;

}

else

{

return keyPath;

}

}

catch (Exception ex)

{

}

return keyPath;

}

// Get s3 bucket configs data:

public async Task<List<AppConfigurations>> GetS3JsonFileData()

{

//var bucketName = "rsasupportapiproject";

//rsasupportapiproject/RSASupportConfigurations/AppConfigurations.json

// rsasupportapi / RSASupportConfigurations / AppConfigurations.json

// rsasupportapidemo/RSASupportConfigurations/AppConfigurations.json

var bucketName = "rsasupportapidemo";

var s3FolderFilePath = "RSASupportConfigurations/AppConfigurations.json";

var validatePath = await ExistedFolderPathKey(bucketName, s3FolderFilePath);

var appconfigurations = new List<AppConfigurations>();

if (validatePath != null)

{

var jsonData = await GetS3BucketObjectDetails(bucketName, s3FolderFilePath);

appconfigurations = JsonConvert.DeserializeObject<List<AppConfigurations>>(jsonData);

if (appconfigurations != null) {

return appconfigurations;

}

}

return null;

}

}

}