Asynchronous Data Retrieval with SQL Stored Procedures in C#

Introduction

Asynchronous programming is essential for ensuring responsiveness and efficient resource utilization, particularly in I/O-bound or high-latency scenarios like database interactions. Leveraging async and await with SQL stored procedures can streamline data retrieval while preserving application performance. Here, I share my typical approach to retrieving data asynchronously using SQLClient with a SQL stored procedure in C#.

Sample Module Code

```
public static async Task<string[]> GetFromEmailAddress(int? storeid)
{
  var config = new ConfigurationBuilder().AddJsonFile("appsettings.json").Build();

  string fromName = string.Empty;
  string fromEmailAddr = string.Empty;
  string? _sql_con_str = configuration["sqlsetting:your_integrated_connection_string"];

  using (var _sql_con = new System.Data.SqlClient.SqlConnection(_sql_con_str))
  {
    // call the stored procedure using SQL Client
    using (var _sql_cmd = new System.Data.SqlClient.SqlCommand("stored_procedure_name", _sql_con))
    {
      _sql_cmd.CommandType = System.Data.CommandType.StoredProcedure;
      _sql_cmd.Parameters.AddWithValue("@p_store_id", storeid);

      await _sql_con.OpenAsync(); // asynchronously open the connection

      using (var _sql_dr = await _sql_cmd.ExecuteReaderAsync()) // execute reader asynchronously
      {
        while (await _sql_dr.ReadAsync()) // asynchronously read rows
        {
          fromName = _sql_dr["FromName"] as string ?? string.Empty;
          fromEmailAddr = _sql_dr["FromEmailAddr"] as string ?? string.Empty;
```

```
        }
      }
    }
  }

  return new[] { fromName, fromEmailAddr };
}
```

Sample of Calling the Module

```
string[] FromAddressInfo = await GetFromEmailAddress(store_id);

// ensure FromAddressInfo is valid and contains at least two values
if (FromAddressInfo == null || FromAddressInfo.Length < 2 ||
  string.IsNullOrWhiteSpace(FromAddressInfo[0]) ||
  string.IsNullOrWhiteSpace(FromAddressInfo[1]))
  {
  return new ContentResult() { Content = "From Address or From Name not found", StatusCode = 422 };
}


Console.WriteLine($"{FromAddressInfo[0]}"); // display value of the string[] array
Console.WriteLine($"{FromAddressInfo[1]}"); // display value of the string[] array
```

Key Points

Purpose of Asynchronous Data Retrieval

- Enables non-blocking operations for database interactions.
- Improves responsiveness, especially in applications with high concurrency.

Setup and Configuration

- Use appsettings.json for connection strings to enhance maintainability.
- Leverage ConfigurationBuilder for configuration management.

Using SQL Client for Asynchronous Operations

- Demonstrates opening a connection with SqlConnection.OpenAsync().
- Showcases executing commands using SqlCommand.ExecuteReaderAsync().
- Iterates through data with SqlDataReader.ReadAsync().

Error Handling and Validation

- Checks for null or incomplete results from the stored procedure.
- Logs errors and returns meaningful responses for failure cases.

Implementation

- Example code retrieves "From Email Address" and "From Name" from a database via a stored procedure using the store_id variable.
- Constructs a dictionary to store and manage the retrieved values.
- Constructs a string[] array to return both values from the method. Using a string[] array is a lightweight alternative when you only need to return a couple of items, and defining a class would be overkill.
- This approach allows for quick, efficient, and concise data handling, especially when the values are not complex enough to justify the creation of a class or struct.

Sample Code Breakdown

- Highlights the importance of parameterized queries for security.
- Demonstrates structuring results for easy use in calling methods.
    - The string[] array approach is employed here to return two simple values (e.g., "From Email Address" and "From Name") with minimal overhead, providing a more straightforward solution in cases where simplicity is prioritized.

Notes on using a string[] array

- At times, simplicity can outweigh complexity, especially when tackling straightforward tasks. When a method needs to return just two values, defining a dedicated class or struct might feel unnecessarily heavy. In such cases, a lightweight structure like a string[] array can offer a quick and effective solution. Although a string[] array is useful for simple, sequential collections of data and is pragmatic when brevity and focus are valued over formality, it does sacrifice the clarity and type safety that a dedicated class or struct provides. The balance between formality (class/struct) and brevity (array) is key to deciding when to use each approach based on the task's complexity.
- This approach is particularly useful in scenarios where:
    - Performance and simplicity are key: A string[] array eliminates the overhead of defining, instantiating, and maintaining a class or struct.
    - Short-term or one-off requirements: For tasks with limited scope or minimal reusability, using a string[] array avoids cluttering the codebase with rarely used data models.
    - Rapid prototyping: During initial development or proof-of-concept phases, this approach allows for faster iterations.

Conclusion

Using async and await with SQL stored procedures enhances the efficiency of database interactions in C#. Following best practices, such as validation, parameterization, and error logging, ensures robust and secure implementations.