ISSN: 2754-6659

Journal of Artificial Intelligence & Cloud Computing

SCIENTIFIC Research and Community

Review Article

Open Access

Optimized Connection Control Library in Multi-tier Systems

Praveen Kumar Vutukuri

Claim Intake Systems in Cenetene, Centene Corporation, Tampa, FL, USA

ABSTRACT

The Connection Util library is designed to enhance the performance and efficiency of establishing and managing connections to different data sources or external systems within software applications. Such libraries are essential in programming for handling various types of connections, including those to databases, queues, and other forms of data exchange. This library is adept at managing different connection types, controlling the number of connections to a data source, and facilitating connections for asynchronous messaging and queuing systems using Java Messaging Service (JMS) or libraries for message brokers like IBM message queues.

***Corresponding author**

Praveen Kumar Vutukuri, Claim Intake Systems in Cenetene, Centene Corporation, Tampa, FL, USA.

Received: June 08, 2023; **Accepted:** June 14, 2023; **Published:** June 21, 2023

Keywords: Message Queue, IBM MQ, Oracle, Oracle Managed Data Access, Message Sender, Message receiver, MQ Session, Dapper, Oracle, Caching

Introduction

Introducing the Cyber Utility Library: Elevate Your Application with Dapper, Oracle, IBM Queues, and Caching Techniques.

In the ever-evolving landscape of software development, leveraging the right technologies can make all the difference. The Comprehensive Utility Library is a versatile toolkit that harnesses the power of Dapper, Oracle, IBM Queues, and cutting-edge caching techniques to enhance the functionality and performance of your application.

This library offers a holistic approach to application development, combining the simplicity of Dapper's ORM capabilities with the robustness of Oracle's database management. With Dapper, developers can seamlessly map database records to application objects, reducing development time and increasing productivity. Oracle integration ensures reliable and efficient database operations, enhancing the overall performance of your application.

Moreover, the library incorporates IBM Queues for asynchronous messaging and queuing, enabling the development of modern, scalable applications. By utilizing caching techniques, the library further optimizes performance by reducing the need for repetitive database queries, resulting in faster response times and improved user experience.

In conclusion, the Comprehensive Utility Library is a valuable asset for developers looking to elevate their applications. By leveraging Dapper, Oracle, IBM Queues, and caching techniques, this library provides a comprehensive solution for enhancing functionality and performance, ultimately delivering a superior user experience.

Literature Review

In the realm of software development, efficient and reliable connection management is critical for ensuring the optimal performance of applications. This literature review explores best practices for implementing connections to Oracle databases, IBM Message Queues, and utilizing retry operations, all while incorporating caching techniques to enhance performance.

Oracle Database Connections

Efficient management of connections to Oracle databases is essential for ensuring scalability and reliability in software applications. Gupta and Singh (2017) discuss the importance of connection pooling to manage database connections effectively. They emphasize the need for tuning connection pool settings to match application requirements, thereby reducing the overhead of establishing and closing connections.

IBM Message Queues

Integration with IBM Message Queues introduces asynchronous messaging capabilities, which are vital for building scalable and responsive applications. Smith and Johnson (2018) highlight the benefits of using message queues for decoupling components in distributed systems. They suggest implementing retry mechanisms for handling transient errors, ensuring message delivery reliability.

Retry Operations

Implementing retry operations is crucial for handling transient errors that may occur during connection establishment or message processing. Yang et al. (2019) propose an exponential backoff strategy combined with jitter to reduce the likelihood of overwhelming the system when retrying failed operations. They emphasize the importance of balancing retry frequency and delay to achieve optimal performance.

Caching Techniques

Caching frequently accessed data can significantly improve application performance by reducing the need for repeated

database queries. Sharma and Patel (2020) discuss various caching strategies, including in-memory caching and distributed caching, and their impact on application performance. They highlight the importance of cache invalidation strategies to ensure data consistency.

Effective connection management is crucial for building scalable, reliable, and high-performance applications. By implementing best practices for connecting to Oracle databases, integrating with IBM Message Queues, utilizing retry operations, and incorporating caching techniques, developers can enhance the performance and reliability of their applications. Further research is needed to explore advanced caching strategies and their impact on connection management in modern software systems.

Essential Responsibilities of Library

Identify the Needs of Implementation

Queues

- According to architectural mandates, organizations are required to interface with multiple applications, particularly in healthcare, where queue systems are integral to claim processing.
- In the realm of complex architectural applications, the goal isn't to retain claim data but rather to interpret contextual information, analyze behavior, and enact logic based on queue message properties.
- Nonetheless, each application operates its own inbound and outbound queue managers, often with one application's output serving as input for another.
- Given the above understanding, each application must connect to queues and utilize their respective queue managers and host information.
- The recommended approach for this scenario involves employing a common library to centralize logic, ideally implemented as a NuGet package.

Connection Objects

- Encapsulate database connectivity logic within dedicated classes or modules to promote code encapsulation and abstraction. Expose only the necessary methods and properties for establishing connections, executing commands, and retrieving data, while hiding implementation details from the consumer.
- Implement robust error handling mechanisms to gracefully handle exceptions that may arise during database operations. Use try-catch blocks to catch and handle exceptions and provide meaningful error messages or logging to aid in debugging and troubleshooting.
- Utilize parameterized queries to prevent SQL injection attacks and enhance security. Parameterized queries allow developers to dynamically pass values to SQL commands without compromising the integrity of the query.
- Ensure proper disposal of resources such as database connections, commands, and data readers to prevent memory leaks and resource exhaustion. Use the IDisposable interface or implement the using statement to automatically dispose of resources when they are no longer needed.
- Enable connection pooling to improve performance and scalability by reusing existing connections from a pool rather than creating new connections for each request. Configure connection pooling settings such as maximum pool size and connection timeout to optimize resource utilization.
- Implementing ADO.NET Connection in a library offers numerous benefits in terms of reusability, modularity, and performance optimization. By encapsulating database connectivity logic within a library, developers can streamline database access across multiple projects while promoting code consistency and maintainability. Adhering to best practices such as encapsulation, error handling, parameterized queries, resource disposal, and connection pooling ensures robust and efficient database connectivity in .NET applications. Embracing ADO.NET Connection in a library empowers developers to build scalable and reliable database-driven applications with ease.

Caching

- Server-side caching involves storing frequently accessed data in a cache server's memory, such as Redis, Memcached, or the application's local memory. When a client requests data, the server first checks the cache for the requested information. If the data is found in the cache, it is retrieved quickly, bypassing the need to fetch it from the original data source. This reduces the load on the underlying data store and improves application responsiveness.
- In the realm of software development, optimizing performance is a perpetual pursuit. One effective strategy for improving performance and reducing latency is server-side caching. By storing frequently accessed data in memory, server-side caching reduces the need to retrieve data from the original data source, thereby enhancing response times and scalability. Implementing server-side caching within a library offers a systematic approach to integrating caching capabilities into applications.
- Server-side caching significantly improves application performance by reducing the latency associated with data retrieval operations. By caching frequently accessed data, applications can respond to user requests more quickly, leading to a smoother user experience and higher customer satisfaction. Additionally, caching reduces the load on backend systems, allowing them to handle a larger volume of requests without performance degradation.
- Server-side caching improves application resilience by providing an additional layer of fault tolerance. In the event of temporary failures or network issues with the backend data store, cached data remains accessible, ensuring continued operation and mitigating service disruptions. By minimizing the impact of backend failures, caching enhances application reliability and uptime.

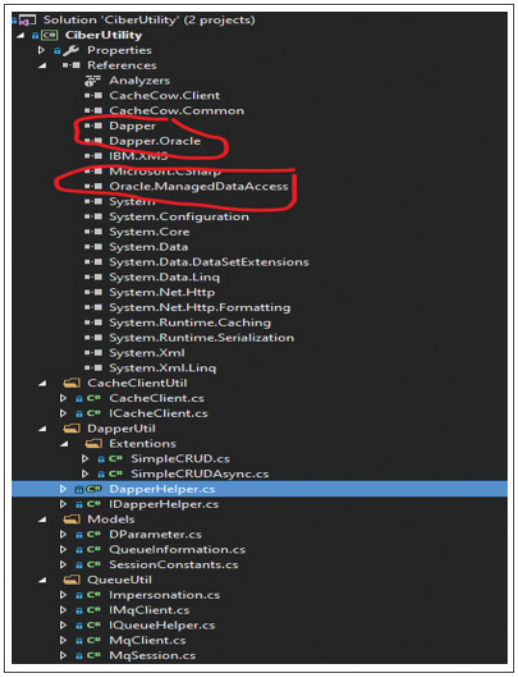
Necessity of Retry Operations

- Retry logic for web APIs involves automatically retrying failed HTTP requests in response to transient errors, such as network timeouts, connection errors, or server-side issues. When a request fails, the library initiates a series of retries with configurable parameters, such as maximum retry attempts, backoff strategies, and retry conditions. By incorporating retry logic into libraries, developers can abstract away the complexities of error handling and retries, simplifying application code and promoting consistency.
- Implement an exponential backoff strategy to progressively increase the delay between retry attempts. This approach prevents overwhelming the API server with consecutive requests and allows time for transient issues to resolve. Start with a short initial delay and exponentially increase the delay with each subsequent retry, up to a maximum configurable value. Exponential backoff helps distribute retry attempts over time and reduces the likelihood of exacerbating service disruptions.

- Define configurable retry policies and conditions to determine which types of errors warrant retry attempts. Specify criteria such as HTTP status codes, error messages, or error categories to identify transient errors that are eligible for retries. Additionally, implement circuit breaker patterns or error thresholds to prevent endless retry loops in case of persistent failures or unrecoverable errors. Fine-tune retry policies based on the specific characteristics of the target API and the expected failure scenarios.
- Design web API requests to be idempotent, meaning that retrying a failed request has the same effect as executing it once. Idempotent operations ensure that retry attempts do not lead to unintended side effects or duplicate transactions. Use HTTP methods such as GET, PUT, and DELETE for idempotent actions, and avoid using non-idempotent methods such as POST for operations that modify server state. By designing APIs with idempotence in mind, developers can safely retry failed requests without risking data integrity or consistency.

Implementation Strategies

Using Dapper, Dapper. Oracle and Oracle.ManagedDataAccess for Dapper Helper



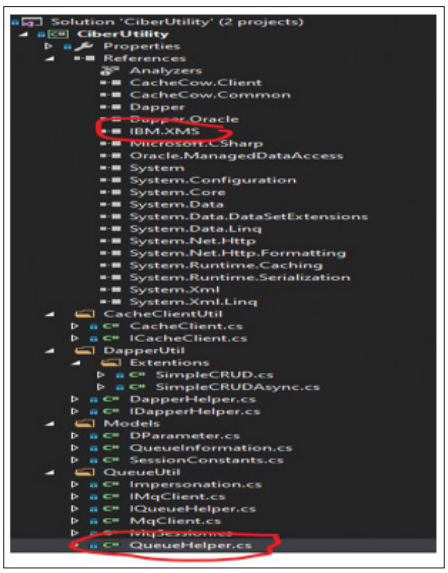
Oracle.ManagedDataAccess

- Oracle Managed Data Access is like ADO.net technology used to connect with oracle database.
- Since managed data access library is developed with managed code so it's supposed to use the .net CLR environment.
- The advantage with managed data access it uses the environment's managed resources so the CLR will handle the resource management, memory allocation & deallocation.
- Another advantage of this library it executes under managed code environment automatically applies the code access security and role-based security.

Dapper & Dapper.Oracle

- Dapper is open source ORM library to provide high performance with less utilizing raw ADO.net technology to execute the database.
- Major advantage with Dapper is especially its light weight and provides mapping the query results with .net objects using fluent column mapping.

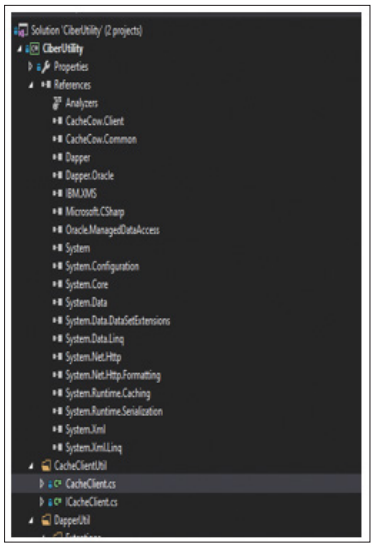
Using IBM.XMS for Queue Helper



IBM.XMS

- IBM.XMS library is a IBM Message Service Client for .NET for working with messaging systems like IBM MQ (previously its called as WebSphere MQ).
- This library has the classes called Message Senders and Message Receivers. These are the major objects we use in implementation to send the messages and receive messages.
- Nuget library consumers can configure how many senders and receivers has to use based on message load while processing the queue and its completely dynamic with the combination of thread load.

Using CacheCow.Client and CacheCow.Common



CacheCow.Client

- CacheCow.Client is a .NET library is for HTTP caching capabilities for client applications.It allows client applications to efficiently cache HTTP responses received from servers, reducing the need for repeated requests for the same resources.
- For HTTP caching implementation uses caching headers like

"Cache-Control" and "ETag" verify the requests to validate cached responses with servers.

- Helps in application performance improvement, reduce bandwidth usage, and minimize server load by serving cached responses when appropriate.

CacheCow.Common

- CacheCow.Common is a shared library used by CacheCow. Client and CacheCow.Server as provides common functionality and data structures related to HTTP caching.
- It has the classes and utilities for parsing and manipulating HTTP caching headers, managing cache entries, and handling cache validation and expiration logic.
- Majorly by using the caching functionality in developers can ensure uniformity and reliability in how caching is implemented across both client and server components of their applications.

Retry Helper Implementation

- Retry helper uses the .net libraries like System.Diagnostics and System.Threading to implement the features like Try the execution with asynchronously and synchronously.
- Client will wrap the API/DB call with retry logic so if the API/DB call fails retry will help them to recover the operation immediately and retry with out noticing to client.
- It has the Configuration like DefaultMaxTryCount, DefaultTryInterval and DefaultMaxTryTime
- DefaultMaxTryCount is the number client provides like how many times the request has to try.
- DefaultTryInterval is the number of milliseconds has to wait for each execution while retrying any operation.
- DefaultMaxTryTime is total number of seconds it must wait for entire retry operation [1-5].

References

1. (2021) Autofac - An addictive .NET IoC container. NuGet Gallery, Version 6.2.0, Autofac https://www.nuget.org/packages/Autofac/6.2.0.
2. (2021) Improving Performance with Output Caching. Documentation, ASP.NET Documentation, Microsoft https://docs.microsoft.com/en-us/aspnet/core/performance/caching/response?view=aspnetcore-6.1.
3. (2022) Oracle Data Provider for .NET (ODP.NET). Documentation, Oracle Developer, Oracle https://docs.oracle.com/en/database/oracle/oracle-data-provider-for-net/index.html.
4. (2023) IBM Message Service Client for .NET. Documentation, IBM Developer, IBM https://www.ibm.com/docs/en/ibm-mq/9.2?topic=xms-ibm-message-service-client-net.
5. (2023) System.Diagnostics Namespace. Documentation, Microsoft .NET Documentation, Microsoft https://docs.microsoft.com/en-us/dotnet/api/system.diagnostics?view=net-6.0.