**Singleton Pattern in Configuration Manager**

**Detailed Problem Statement**

**Background and Context**

"SecureWeb Apps" is a company that develops enterprise-level web applications, such as e-commerce platforms, banking portals, and content management systems, which handle sensitive data and require robust configuration management. These applications rely on a variety of configuration settings, including database connection strings, API keys, authentication tokens, logging levels, feature flags, and environment-specific variables (e.g., development vs. production). These settings are typically loaded from external sources like configuration files (e.g., appsettings.json, XML), environment variables, or a database at application startup to avoid hardcoding sensitive information and to ensure flexibility across deployment environments.

In a web application, especially one running on a multi-threaded server (e.g., ASP.NET Core handling concurrent HTTP requests), managing configurations poses several challenges:

* **Consistency**: Multiple instances of a configuration manager could load settings independently, leading to inconsistent or outdated data across threads, causing unpredictable behavior (e.g., one thread using a stale database connection string).
* **Resource Efficiency**: Repeatedly reading configuration files or querying a database for the same settings wastes CPU, memory, and I/O resources, especially under high traffic conditions.
* **Security Risks**: Multiple instances increase the risk of unauthorized access or modification of sensitive data, such as API keys, if not properly synchronized.
* **Maintenance Complexity**: Without a single, centralized access point, debugging configuration-related issues becomes difficult, as changes in one part of the application may not propagate globally.

To address these challenges, the system requires a design that ensures exactly one instance of the configuration manager exists throughout the application's lifecycle. This instance must serve as the single source of truth for all configuration settings, provide global access to all parts of the application, and ensure thread-safety to handle concurrent access in a multi-threaded environment. The Singleton Pattern is ideal for this scenario, as it restricts instantiation to a single object, supports lazy initialization, and provides mechanisms for thread-safe access, aligning with common practices in .NET applications where services like IConfiguration are often registered as singletons in dependency injection containers.

**Requirements**

1. **System Architecture**:
   * Implement a ConfigurationManager class as the singleton, responsible for storing and managing configuration settings.
   * Use a private Dictionary<string, string> to store key-value pairs of settings (e.g., "DatabaseConnection" = "Server=localhost;Database=test;" and "ApiKey" = "12345-abcde").
   * Ensure the class has a private constructor to prevent external instantiation, and provide a static property Instance to access the single instance.
   * Implement thread-safety using the double-checked locking pattern with a lock object to ensure safe initialization in a multi-threaded environment, such as a web server handling concurrent requests.
2. **Configuration Loading and Access**:
   * Include a private method LoadSettings() that simulates loading configuration data from an external source (e.g., a file or database). For this case study, hardcode sample settings to keep the implementation simple:
     + "DatabaseConnection" = "Server=localhost;Database=test;"
     + "ApiKey" = "12345-abcde"
   * Provide a public method GetSetting(string key) that retrieves the value for a given key. If the key is not found, throw a KeyNotFoundException with a descriptive message.
   * Optionally, include a method like SetSetting(string key, string value) to update settings, ensuring thread-safety (e.g., using locks) if modifications are allowed. For simplicity, this case study focuses on read-only access after initial loading.
   * The LoadSettings() method should be called only once during the singleton's initialization to avoid redundant operations.
3. **Singleton Implementation Details**:
   * Use **lazy initialization** to create the singleton instance only when first accessed via the Instance property, reducing startup overhead.
   * Ensure **thread-safety** using double-checked locking: check if \_instance is null before and after acquiring a lock to prevent multiple threads from creating separate instances.
   * Use a private static readonly object for the lock to synchronize access during initialization.
   * Prevent external instantiation or inheritance by making the constructor private and optionally sealing the class to avoid subclassing issues.
4. **Client Usage and Demonstration**:
   * In a main program (e.g., a console application), demonstrate the singleton's functionality by:
     + Accessing the ConfigurationManager instance from the main thread and retrieving a setting (e.g., "DatabaseConnection").
     + Simulating concurrent access by creating a separate thread (using System.Threading.Thread) that accesses the same ConfigurationManager instance and retrieves another setting (e.g., "ApiKey").
     + Verifying that both threads access the same instance by comparing references (e.g., config1 == config2 should return true).
     + Displaying the retrieved settings and the result of the instance comparison.
   * Include error handling to catch and display exceptions, such as attempting to retrieve a non-existent key.
   * The client code should interact only with the ConfigurationManager.Instance property and its public methods, avoiding direct access to the internal dictionary.
5. **Non-Functional Requirements**:
   * **Scalability**: The design should support future extensions, such as loading settings from a file or database, without altering the singleton's core structure.
   * **Security**: While not the focus, the implementation should assume sensitive data (e.g., API keys) is stored securely, with read-only access for simplicity. If modifications are added, they should be protected against concurrent writes.
   * **Maintainability**: Follow C# best practices, including clear naming conventions (e.g., ConfigurationManager, GetSetting), proper encapsulation, and minimal dependencies (use only System, System.Collections.Generic, and System.Threading namespaces).
   * **Error Handling**: Handle invalid inputs (e.g., null or non-existent keys) gracefully with appropriate exceptions and messages.

**Expected Outcomes**

* The application should output configuration settings retrieved from the singleton instance, such as the database connection string and API key, from both the main thread and a simulated concurrent thread.
* The demonstration should confirm that the same instance is accessed across threads, ensuring consistency.
* Error handling should gracefully manage invalid key requests, providing clear feedback.
* This implementation mirrors real-world configuration management in frameworks like ASP.NET Core, where a singleton IConfiguration service provides global access to settings, making it a practical case study for learning.