# LOGICAL AND REVIEW CODE

**AND**

# BUILD WEB APP TEST BASED ON USE STORY

LEMBAGA PEMJAMIN SIMPANAN 2023

GROUP SISTEM INFORMASI DIVISI PENGEMBANGAN APLIKASI

This file and its contents and format are the property of the Deposit Insurance Corporation (IDIC). It is not permitted to be distributed, duplicated and/or used in part or in whole in any form without prior written permission from LPS except for the purposes of carrying out this work.

## Goals:

### This text is intended to measure the ability to write coding as well as measure the ability to carry out coding reviews.

Another technical measurement the candidate should be proven in web application development using clean architecture pattern.

## Task:

### Please re-write the code, doing coding, analysing and fixing the code! (question number 1 to 7). Crate a solution using Microsoft Visual Studio and store all source code to github.com and create readme.md to explain your opinion!

* For number 8 read user story carefully and doing the task, and do not forget to store the code in github.com

## Logical and Review Code

* 1. **How about your opinion..?**

if (application != null)

{

if (application.protected != null)

{

return application.protected.shieldLastRun;

}

}

**Key**:

cleaner and easier to read code.

**Answer :**

However, you can make it more concise and easier to read using the null conditional operator (?.) and the null-coalescing operator (??). Here's a refined version:

In Code C#

return application?.protected?.shieldLastRun ?? defaultValue;

In this version:

1. application?.protected?.shieldLastRun safely navigates through the properties. If any of the properties are null, it will return null instead of throwing an exception.
2. Replace defaultValue with whatever default you want to return if shieldLastRun is null.

If you want to return a specific type (like DateTime, for example), make sure defaultValue matches that type.

* 1. **How about your opinion.**

public ApplicationInfo GetInfo()

{

var application = new ApplicationInfo

{

Path = "C:/apps/", Name = "Shield.exe"

};

return application;

}

**Key**:

return more than one value from a class method.

**Answer:**

If you want to return a dynamic set of values, a dictionary might be useful.

public Dictionary<string, string> GetInfo()

{

return new Dictionary<string, string>

{

{ "Path", "C:/apps/" },

{ "Name", "Shield.exe" }

};

}

var infoDict = GetInfo();

Console.WriteLine($"Path: {infoDict["Path"]}, Name: {infoDict["Name"]}");

* 1. **How about your opinion.**

class Laptop

{

public string Os{ get; set; } // can be modified public Laptop(string os)

{

Os= os;

}

}

var laptop = new Laptop("macOs"); Console.WriteLine(Laptop.Os); // Laptop os: macOs

**Key**:

modifications by using private members.

class Laptop

{

private string os; // Private member

public string Os // Public property

{

get { return os; }

private set { os = value; }

}

public Laptop(string os)

{

Os = os; // Set the property via the constructor

}

}

var laptop = new Laptop("macOS");

Console.WriteLine($"Laptop OS: {laptop.Os}"); // Use the instance to access the property

1. **Private Field**: The os member is now private.
2. **Property Access**: The setter for Os is private, preventing modification from outside the class.
3. **String Interpolation**: Used string interpolation for cleaner output.
4. **Correct Instance Usage**: Accessed Os through the laptop instance instead of the class itself.
   1. **How about your opinion.**

using System;

using System.Collections.Generic;

namespace MemoryLeakExample

{

class Program

{

static void Main(string[] args)

{

var myList = new List(); while (true)

{

// populate list with 1000 integers for (int i = 0; i < 1000; i++)

{

myList.Add(new Product(Guid.NewGuid().ToString(), i));

}

// do something with the list object Console.WriteLine(myList.Count);

}

}

}

class Product

{

public Product(string sku, decimal price)

{

SKU = sku; Price = price;

}

public string SKU { get; set; } public decimal Price { get; set; }

}

}

**Key**:

Keeping references to objects unnecessarily

### **Changes:**

1. **Clearing the List**: After processing, myList.Clear() is called to remove all elements, preventing memory accumulation.
2. **Formatted Output**: Used string interpolation for the output for better readability.

### **Considerations:**

* If you intend to keep the objects in memory for some reason, consider using a different approach (e.g., using a fixed-size collection or storing only the latest entries).
* Be cautious with infinite loops; ensure you have a break condition for production code to avoid unintentional resource exhaustion.

using System;

using System.Collections.Generic;

namespace MemoryLeakExample

{

class Program

{

static void Main(string[] args)

{

var myList = new List<Product>();

while (true)

{

// Populate list with 1000 integers

for (int i = 0; i < 1000; i++)

{

myList.Add(new Product(Guid.NewGuid().ToString(), i));

}

// Do something with the list object

Console.WriteLine($"Count: {myList.Count}");

// Clear the list to free up memory

myList.Clear();

}

}

}

class Product

{

public Product(string sku, decimal price)

{

SKU = sku;

Price = price;

}

public string SKU { get; set; }

public decimal Price { get; set; }

}

}

* 1. **How about your opinion.**

using System;

namespace MemoryLeakExample

{

class Program

{

static void Main(string[] args)

{

var publisher = new EventPublisher();

while (true)

{

var subscriber = new EventSubscriber(publisher);

// do something with the publisher and subscriber objects

}

}

class EventPublisher

{

public event EventHandler MyEvent;

public void RaiseEvent()

{

MyEvent?.Invoke(this, EventArgs.Empty);

}

}

class EventSubscriber

{

public EventSubscriber(EventPublisher publisher)

{

publisher.MyEvent += OnMyEvent;

}

private void OnMyEvent(object sender, EventArgs e)

{

Console.WriteLine("MyEvent raised");

}

}

}

}

**Key**:

event handlers

**Changes:**

1. **Single Subscriber Creation**: Instead of creating a new subscriber in an infinite loop, we create one and use it to handle events.
2. **Implementing IDisposable**: The EventSubscriber class implements IDisposable, allowing it to unsubscribe from events cleanly when it is no longer needed.
3. **Simulated Events**: A simple loop simulates raising events a few times before disposing of the subscriber.

### **Considerations:**

* This approach minimizes the risk of memory leaks by ensuring that subscriptions are managed properly.
* Depending on your application’s needs, you might want to create and destroy subscribers conditionally rather than continuously.

|  |
| --- |
| **using System;**  **namespace MemoryLeakExample**  **{**  **class Program**  **{**  **static void Main(string[] args)**  **{**  **var publisher = new EventPublisher();**  **// Create a single subscriber**  **var subscriber = new EventSubscriber(publisher);**  **// Simulate raising events**  **for (int i = 0; i < 10; i++)**  **{**  **publisher.RaiseEvent();**  **}**  **// Clean up**  **subscriber.Dispose();**  **}**  **}**  **class EventPublisher**  **{**  **public event EventHandler MyEvent;**  **public void RaiseEvent()**  **{**  **MyEvent?.Invoke(this, EventArgs.Empty);**  **}**  **}**  **class EventSubscriber : IDisposable**  **{**  **private readonly EventPublisher publisher;**  **public EventSubscriber(EventPublisher publisher)**  **{**  **this.publisher = publisher;**  **publisher.MyEvent += OnMyEvent;**  **}**  **private void OnMyEvent(object sender, EventArgs e)**  **{**  **Console.WriteLine("MyEvent raised");**  **}**  **public void Dispose()**  **{**  **// Unsubscribe from the event to prevent memory leaks**  **publisher.MyEvent -= OnMyEvent;**  **}**  **}**  **}** |

* 1. **How about your opinion.**

using System;

using System.Collections.Generic; namespace MemoryLeakExample

{

class Program

{

static void Main(string[] args)

{

var rootNode = new TreeNode(); while (true)

{

// create a new subtree of 10000 nodes var newNode = new TreeNode();

for (int i = 0; i < 10000; i++)

{

var childNode = new TreeNode(); newNode.AddChild(childNode);

}

rootNode.AddChild(newNode);

}

}

}

class TreeNode

{

private readonly List<TreeNode> \_children = new List<TreeNode>(); public void AddChild(TreeNode child)

{

\_children.Add(child);

}

}

}

**Key**:

Large object graphs

**Changes:**

1. **ChildCount Property**: Added a property to get the count of child nodes for easier monitoring.
2. **ClearChildren Method**: Implemented a method to clear the children of the TreeNode, freeing up memory after processing.
3. **Processing Simulation**: Added a print statement to simulate processing the tree.

### Considerations:

* This approach prevents the continuous growth of the tree structure, allowing your application to run for longer without consuming excessive memory.
* Depending on your application's requirements, consider a more sophisticated memory management strategy, like using a pooling mechanism for nodes if they need to be reused.

|  |
| --- |
| using System;  using System.Collections.Generic;  namespace MemoryLeakExample  {  class Program  {  static void Main(string[] args)  {  var rootNode = new TreeNode();  while (true)  {  // Create a new subtree of 10,000 nodes  var newNode = new TreeNode();  for (int i = 0; i < 10000; i++)  {  var childNode = new TreeNode();  newNode.AddChild(childNode);  }  rootNode.AddChild(newNode);  // Simulate processing the tree here  Console.WriteLine($"Root has {rootNode.ChildCount} children");  // Optionally clear children to prevent memory growth  rootNode.ClearChildren();  }  }  }  class TreeNode  {  private readonly List<TreeNode> \_children = new List<TreeNode>();  public void AddChild(TreeNode child)  {  \_children.Add(child);  }  public int ChildCount => \_children.Count;  public void ClearChildren()  {  \_children.Clear(); // Clear children to free memory  }  }  } |

* 1. **How about your opinion.**

using System;

using System.Collections.Generic; class Cache

{

private static Dictionary<int, object> \_cache = new Dictionary<int, object>();

public static void Add(int key, object value)

{

\_cache.Add(key, value);

}

public static object Get(int key)

{

return \_cache[key];

}

}

class Program

{

static void Main(string[] args)

{

for (int i = 0; i < 1000000; i++)

{

Cache.Add(i, new object());

}

Console.WriteLine("Cache populated"); Console.ReadLine();

}

}

**Key**:

Improper caching

### **Changes**

1. **Concurrent Dictionary**: Using ConcurrentDictionary ensures thread safety, allowing multiple threads to read and write without issues.
2. **TryGetValue**: This method prevents exceptions when trying to access a non-existent key.
3. **Basic Cache Management**: Added methods for removing entries and clearing the cache.

### **Considerations**

### **Eviction Policy**: Depending on your application's needs, consider implementing an eviction policy (e.g., Least Recently Used, Time-to-Live) to manage memory usage effectively.

### **Generic Type Support**: If you need type safety, you can make Cache a generic class. This would require adjusting the methods to handle type parameters.

### **Logging and Monitoring**: Implement logging to track cache hits and misses for performance tuning.

|  |
| --- |
| using System;  using System.Collections.Concurrent;  class Cache  {  private static ConcurrentDictionary<int, object> \_cache = new ConcurrentDictionary<int, object>();  public static void Add(int key, object value)  {  \_cache[key] = value; // This will update the value if the key already exists.  }  public static object Get(int key)  {  \_cache.TryGetValue(key, out var value); // Safe retrieval  return value;  }  public static void Remove(int key)  {  \_cache.TryRemove(key, out \_); // Safely remove the key  }  public static void Clear()  {  \_cache.Clear(); // Clear all entries if needed  }  }  class Program  {  static void Main(string[] args)  {  for (int i = 0; i < 1000000; i++)  {  Cache.Add(i, new object());  }  Console.WriteLine("Cache populated");  Console.ReadLine();  }  } |