Resource Clean Up in .NET

It can be surprisingly confusing to get a clear idea of all the facts and options regarding resource clean up in .NET and how to use it in N-Tier web applications with your data access or other classes. In this document I attempt to provide an overview of the garbage collection, the Dispose pattern and how you can practically use it in an MVC application.

Garbage Collection Basics

.NET code is managed code, which means that the .NET CLR is responsible for executing your code and for providing several services to you. One of them is Memory Management through Garbage Collection. To understand how garbage collection works it is necessary to understand a few basic things about resource allocation in .NET.

Every class you create and every existing type in .NET uses resources. Most of your classes will use the most commonly used resource, computer memory. However you may write classes or use .NET classes that use other resources which may be a bit more involved such as a database connection. A database connection is ultimately a TCP connection between client (your application) and server (your database server) and needs to be closed when no longer needed. To close a TCP connection, open or close a file etc. the .NET CLR needs to communicate with the Operating System. When that happens, it is pretty much obvious that the CLR stops being in control of things. The CLR is "smart enough" to free memory for you when it is no longer needed but it is not going to do anything to close an open database connection for you, this is your responsibility. So to put all this together, there are 2 types of resources

a) Managed Resources – that is the memory used by your & .NET's types – CLR Control
b) Unmanaged Resources – file handles, database connection etc. – Outside CLR Control

It is important to understand the difference between managed and unmanaged resources because that is what dictates your "Resource Clean Up" policy. Let's provide a summary here before we go into more detail.

Resource Clean Up Policy

Managed Resources

- No clean up code required (very rare & exotic exceptions to that rule). The Garbage Collector will automatically do this for you and since it knows what it's doing it is better not to try to mess with it or try to explicitly call it by using GC.Collect()

Unmanaged Resources

- (Not Deterministic) Provide a finalize method. This method performs the clean up (e.g. close a database connection) when the Garbage Collection runs and just before the memory is reclaimed
- (Deterministic) Implement the IDisposable interface and allow Dispose to be called on your types. The resource cleanup will happen when Dispose is called ***BUT*** the memory will still be reclaimed by the Garbage Collector

As you can see things are very straightforward for managed resources, there is not much for you to do. For unmanaged resources, you have 2 options, deterministic and nondeterministic clean up. It is quite obvious that deterministic clean up should be used when the following statements are true

- You know for sure that your object is no longer needed in your application and you can safely dispose of it

- The resource used is an “expensive” one and needs to be closed as soon as possible.

To describe this from a different perspective, let’s see ***when*** clean up can possibly be performed starting from the earliest possible moment in time.

- When Dispose is called (either directly or by using the ***“using”*** pattern)
- When Garbage Collection is performed and before memory is reclaimed (if a finalize method is provided)
- When the process terminates (in that case the OS will reclaim all resources)

One important detail to keep in mind is the difference in timing between resource clean up and memory deallocation. Memory deallocation (we are always talking about managed memory allocated by using the new keyword) is ***ALWAYS*** the responsibility of the Garbage Collector and you have nothing to do with it. The memory an object uses will always be deallocated at garbage collection, the resources it uses however may be cleaned up before that time as already explained.

Now that we have the basics clear let’s have a look at a few details

Garbage Collection Internals

Object Allocation

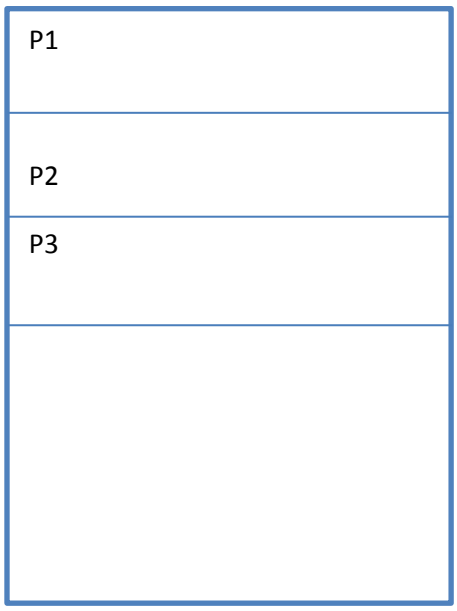
As you probably already know, objects in .NET are allocated by the CLR in a managed section of process memory called the managed heap. The managed heap is a contiguous block of memory to which new objects are added and from which no longer needed objects are removed. When you call something like

Person p1 = new Person(“John”);

The following things take place

- CLR calculates how much memory is needed by your new object and if there is enough your object is created in the heap (if there is not enough memory the CLR will run the Garbage Collector to free some memory, if there is still no memory after this an OutOfMemoryException is thrown)
- The new keyword returns to you the memory address of the newly allocated object which is the reference that p1 holds

Managed Heap



Person p1 = new Person(“John”);

Person p2 = new Person(“Peter”);

Person p3 = new Person(“Tim”);

Garbage Collection

When the garbage collector runs it will check the heap trying to find those objects that are no longer "referenced" and therefore no longer used by the application and reclaim their memory. The garbage collector will try to not only free memory but to also prevent memory fragmentation by compacting memory used by nongarbage objects. So if P2 was to be deallocated, P3 will be moved to be next to P1 in place of P2. All that responsibility makes the garbage collector an expensive and complicated operation and that is why you should avoid using something like GC.Collect to force it to run unless you really know what you are doing.

When does the Garbage Collector run?

The simplest explanation is that the Garbage Collector runs when there is no more available memory on the Heap. That however is an oversimplification. There are a lot of details in how the Garbage Collector does what it does and you can find more details on this if you wish but for your day to day programming tasks the most important concept to keep in your mind is that garbage collections are not deterministic. You cannot possibly predict when the next garbage collection will happen so you are better off not counting on it if you have expensive resources that you need to clean.

Finalize Methods

The most important idea to keep in mind here is that Finalize methods are useful to you if you are doing ***your own*** management of unmanaged, native resources. This is an advanced and not so common scenario so I am not going to cover it in any detail. The most common scenario is that you simply use or wrap an existing .NET class that uses a native resource such as a FileStream or a DbContext object. These classes already implement IDisposable for you so you should simply use the Dispose method they provide.

For the sake of completeness let's see the basics of a Finalize method. Since a Finalize method is kind of a special method it has special syntax.

```
public MyClass
{
        ….

        ~MyClass()
        {
                //call some OS API function here or do whatever you may need to do to  clean up!
                …
        }
}
```

As you can see this method starts with a tilde and it usually needs to interact at a low level with the OS to perform some clean up operation. The Finalize method ***can only be invoked by the garbage collector and not by your code***.

Finalize methods will be called when the garbage collector runs and that includes situations such as Application Domain unloading and process termination.

The Dispose Pattern

If your classes or the classes of .NET use or wrap around a class that uses native resources you usually need to have a way to deterministically handle resource clean up. To do this you implement the IDisposable interface. Implementing the IDisposable interface says 2 important things to the users of your type.

- Your class uses (directly or indirectly) some unmanaged resource that needs to be cleaned up asap by calling Dispose
- Calling code can use the “using” pattern to make it easier to write code that disposes your type.

Let us see a practical implementation of Dispose in a Repository class that uses Entity Framework (for Ling to Sql the basic idea would be pretty much the same)

```
public class BaseRepository : IDisposable
{
    protected EShopData dbContext;

    protected BaseRepository(EShopData db)
    {
        this.dbContext = db;
    }

    //MORE CODE HERE…

    //DISPOSING CODE
    private bool disposed = false;
    private void Dispose(bool disposing)
    {
        if (!this.disposed)
        {
            if (disposing)
            {
                dbContext.Dispose();
            }
        }
        this.disposed = true;
    }

    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
}
```

Let me just say that the code above is “ideal” and not entirely necessary. It could be stripped down to something as simple as

```
    public void Dispose()
    {
          dbContext.Dispose();
    }
```

The longer version however demonstrates a few basic principles about Disposing objects that you should have in mind so it may be of benefit to get used of this syntax as an easy for you to remember the “big picture” of disposing and resource clean up. If you were to put some comments around this code to explain to yourself what is happening the code would look something like this.

```
//DISPOSING CODE FOR MYCLASS
 private IntPtr handle;
  …

 private bool disposed = false;
 private void Dispose(bool disposing)
 {
     //If object not already disposed…
     if (!this.disposed)
     {
```

```
            // if the object is being disposed by the programmer not the garbage collector
            // safe to call methods on objects of this class since their finalize methods could
            // not possibly have executed
            if (disposing)
            {
                //dispose managed resources
                dbContext.Dispose();
            }
            //dispose any unmanaged resources
            CloseHandle(IntPtr);
            Handle = IntPtr.Zero;
        }
        //mark object as disposed
        this.disposed = true;
    }

    public void Dispose()
    {
        Dispose(true);

        //since this method is going to dispose the object there is no need for the Finalize
        //method to execute during garbage collection
        GC.SuppressFinalize(this);
    }

    ~MyClass()
     {
          Dispose(false);
     }


     public void Close()
     {
          Dispose();
     }
```

This code is here just for demonstration purposes, it is rather unlikely that you will come across code that uses a dbContext and an IntPtr in real life. As you can see Dispose(bool disposing) is the method responsible for the clean up. Instead of duplicating logic and code all other methods use it. Dispose() is the actual implementation of the IDisposable interface. The Finalize method ensures that if Dispose is not called programmatically, it will be called when the object is garbage collected. The Close() method is simply a convenience method for those who find Close conceptually a more appropriate name for the resource clean up they perform.

Using keyword

Between the time your type is instantiated and the time Dispose is called on it an exception may be thrown that prevent the call to Dispose from ever taking place. To address this problem you can use a try/finally block or simply use the using keyword.

```
try
{
        MyClass a = new MyClass();
        …
}
finally
{
        a.Dispose()
}
```

OR

```
using (var a = new MyClass())
{
        ….
}
```

If you use “using” which is just a convenient syntax, the compiler will produce the try/finally block for you.

Linq to Sql and Entity Framework

The DataContext and DbContext/ObjectContext classes of Linq to Sql and Entity Framework implement IDisposable. However it is also claimed that these classes are very good at managing the database connections for you. They create sql queries, send them to the database, retrieve the results and then close the connection. Thus, it is not really that important for you to impement the Dispose pattern in your repositories or other classes that use them just to clean them up. There is a lot of confusion out there and it is quite annoying that it is not always easy to find crystal clear answers to such basic questions. There are those who claim that you shouldn’t really call Dispose explicitly on these classes, you just write more code that is not really necessary and which adds little real value and there are those who think that calling Dispose is what the creator of the class wanted us to do and that is why the type implements IDisposable in the first place. While we all know that practical application can sometimes be light years away from theory I am a supporter of the second opinion. If we need to start guessing or have to look at the esoteric details of each .NET type we use to see if it automagically manages some resource for us despite implementing the IDisposable interface then we may well need to spend a lifetime on this task. I prefer the clarity and conceptual integrity of the second approach where you take responsibility for cleaning up these resources yourself. Not only it is educational but it can also be considered good “defensive programming” to do so to ensure that your resources are cleaned up and that you get into the right mind set when it comes to structuring your code.

Resource Clean Up in MVC and Web Forms

Web Forms

Depending on your system architecture, you may be using a data access class, whether that is a DAO class or a repository does not matter that much, what does matter is if you have implemented the IDisposable interface or not. If you haven’t then you depend on DataContext’s or DbContext’s ability to manage connections and these objects (assuming that you don’t make the mistake to make them static or use the singleton pattern which are both the wrong approaches for this) will be cleaned up when the page unloads. If you have implemented IDisposable and you are looking for the best place to put it then you can use the Unload event.

“The Unload event is raised after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded and cleanup is performed.”

ASP.NET MVC

The Controller class of MVC from which your own Controllers inherit implements IDisposable so all you have to do is override this method. When a request comes in in MVC, a controller factory creates a Controller object to handle your request. After your controller does its stuff and the view is rendered, the controller factory can now cleanup the controller by calling its Dispose method. So whether you have a DbContext directly in your controller or you have a repository class you can simply write something like

```
protected override void Dispose(bool disposing)
{
        _orderRepository.Dispose();
        //OR if you are using a DbContext directly
        //dbContext.Dispose();
        Base.Dispose(disposing);
}
```

According to MSDN

"Developers rarely have to override Dispose. When you override Dispose, override the Dispose method, which takes a Boolean parameter."

I am not sure why the requirement for calling the overload that uses a boolean, that will probably become obvious if you delve into the mvc source code and see how the Controller Factory is handling the disposal of controllers but since this is not really a matter of great importance I ll leave it as an exercise for some other day. Hopefully this document explained in some detail and with the required clarity all that you need to know for "day to day" cleanup tasks in your web application code! Let's summarize again the most important concepts.

Managed vs Unmanaged Resources

a) Managed Resources – that is the memory used by your & .NET's types – CLR Control
b) Unmanaged Resources – file handles, database connection etc. – Outside CLR Control

Managed Resources Cleanup

- No clean up code required (very rare & exotic exceptions to that rule). The Garbage Collector will automatically do this for you and since it knows what it's doing it is better not to try to mess with it or try to explicitly call it by using GC.Collect()

Unmanaged Resources Cleanup

- (Not Deterministic) Provide a finalize method. This method performs the clean up (e.g. close a database connection) when the Garbage Collection runs and just before the memory is reclaimed
- (Deterministic) Implement the IDisposable interface and allow Dispose to be called on your types. The resource cleanup will happen when Dispose is called ***BUT*** the memory will still be reclaimed by the Garbage Collector

When to use IDisposable

- You know for sure that your object is no longer needed in your application and you can safely dispose of it
- The resource used is an "expensive" one and needs to be closed as soon as possible.

Cleanup Timeline (starting from the earliest point in time)

- When Dispose is called (either directly or by using the ***"using"*** pattern)
- When Garbage Collection is performed and before memory is reclaimed (if a finalize method is provided)
- When the process terminates (in that case the OS will reclaim all resources)