**Contents**

[Resource Utilization 2](#_Toc222201551)

[Releasing of finished resources 2](#_Toc222201552)

[String concatenation 2](#_Toc222201553)

[String comparison 3](#_Toc222201554)

[ArrayList, HashTables, Dictionary 3](#_Toc222201555)

[Performance 4](#_Toc222201556)

[ForEach, ForLoop, List.ForEach 4](#_Toc222201557)

[Multiple property calls 4](#_Toc222201558)

[Exit conditions 5](#_Toc222201559)

[Web Application Performance 5](#_Toc222201560)

[Application State 5](#_Toc222201561)

[Caching 6](#_Toc222201562)

[Session State 6](#_Toc222201563)

[ViewState 7](#_Toc222201564)

[Extensions 7](#_Toc222201565)

[Data Access 8](#_Toc222201566)

[LINQ (Language INtegrated Query) 8](#_Toc222201567)

**Resource Utilization**

**Releasing of finished resources**

Cleaning up of resources used by your code is important to ensure that there are always sufficient resources to service your other requests. Ensure that you call the **dispose()** or **close()** function for all disposable objects.

1. Other than having try.. catch, add in **finally** to explicitly dispose of any objects which you do not use

Example of try..catch..finally

SqlDataSource DS = new SqlDataSource();

try

{

DS.Select(DataSourceSelectArguments.Empty);

}

catch

{

}

finally

{

DS.Dispose();

}

1. Or better yet, use the **using** keyword, which will always call the dispose() function of the object even in the event of exceptions.

Example of **using**

using (SqlDataSource DS = new SqlDataSource())

{

DS.Select(DataSourceSelectArguments.Empty);

}

1. ASP.NET Pages have an onunload event which you can use to explicitly call dispose(). This event will be called even if an exception occurred anywhere during the processing of the page. Always remember to include base.OnUnload(e) to allow the page to do its own unloading processes.

Example of **onunload**

protected override void OnUnload(EventArgs e)

{

CustomClass.Dispose();

base.OnUnload(e);

}

**String concatenation**

String concatenation is a very expensive process, use **StringBuilder** instead, you will get a very great performance improvement.

System.Text.StringBuilder sb = new System.Text.StringBuilder();

sb.Append("aa");

sb.Append("bb");

sb.ToString();

**String comparison**

Instead of doing case insensitive string comparisons using

string1.toLower() == string2.toLower()

use the following string function

*String.Compare (string strA, string strB, bool ignoreCase);*

String.Compare(string1, string2, true);

**toLower()** functions involve temporary string allocations, which can be very expensive.

**ArrayList, HashTables, Dictionary**

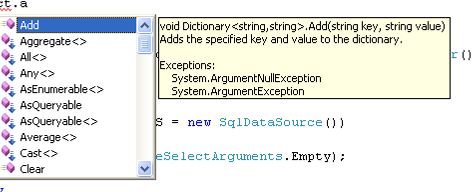
Although ArrayList is very flexible, it is extremely resource intensive, especially when you are storing primitive datatypes. ASP.NET 2.0 introduces new Generic objects which you can use instead. These Generics are much better at handling lists compared to ArrayList. Even better, the functions for these Generic objects are typed, meaning you do not need to worry about stray objects going into the list.

If you are doing random access to the values, you can use the Sorted equivalents (**SortedList, SortedDictionary**). SortedDictionary implements a red-black tree and gives you O(log n) retrieval time, compared to O(n) for Dictionary.

Using System.Collections.Generic

List<string> list = new List<string>;

Dictionary<string, string> dict = new Dictionary<string, string>();



**Performance**

**ForEach, ForLoop, List.ForEach**

Although using ForEach is good as it gives strong typing, it is actually slower than using a normal for loop, but only if the for loop condition refers to a direct primitive value (e.g integer of value 10). If for example you are iterating thru a List<> object, the fastest code will be

int ttl = list.Count();

for (int j = 0; j < ttl; j++)

{

string y = list[j];

}

followed by

foreach (string s in list)

{

string y = s;

}

And last place is

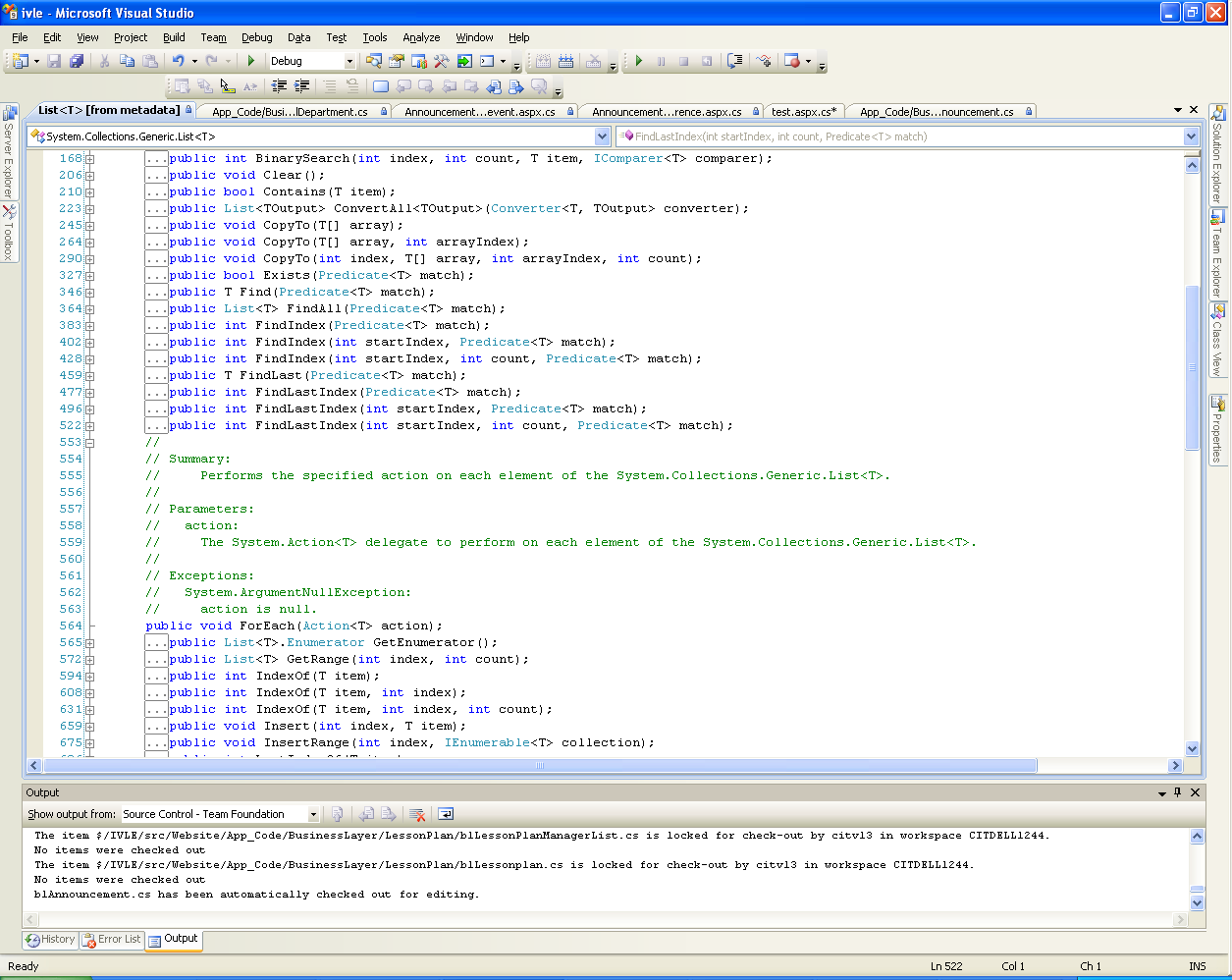
for (int j = 0; j < list.count(); j++)

{

string y = list[j];

}

If you are using the generics list class, it comes with an additional ForEach function which gives a much better performance compared to using the methods listed above.



list.ForEach(delegate(string s) { string y = s; });

**Multiple property calls**

If you frequently call a property, consider storing it in a variable. Property calls are expensive compared to a lookup for a variable. As an example just look at the following 2 examples, the second sample where the count() property is called during each iteration dramatically increases the runtime by at least threefolds.

//Storing the count in a variable and using the variable in the forloop

int ttl = list.Count();

for (int j = 0; j < ttl; j++)

{

string y = list[j];

}

//Referencing the count property each time the forloop is run

for (int j = 0; j < list.count(); j++)

{

string y = list[j];

}

**Exit conditions**

If you have multiple exit conditions, put them when it is first possible to check and also put the one most likely to happen right on top. This reduces the amount of checks needed to exit, which can give you a performance improvement.

So instead of

bool isValid(string UserID, string Password)

{

int result = executesql("select count(\*) from users where userid=... and password=...");

if (result == 1)

return true;

}

Consider the following

bool isValid(string UserID, string Password)

{

//ensure UserID contains something

if (UserID == null || UserID.Length < 1)

return false;

//ensure password contains something

if (Password == null || Password.Length < 1)

return false;

int result = executesql("select count(\*) from users where userid=... and password=...");

if (result == 1)

return true;

}

This reduces the expensive db call if userid or password is empty.

**Web Application Performance**

**Application State**

Each time a web application starts, it creates an application state dictionary which you can use to store values pertaining to the application. For example you can store the application name inside application state for access by your aspx pages. Note that items stored in application exist for the entire duration when the application is available. As long as you initialize the value, you do not really need to check when accessing the parameter to see if it is null.

e.g inside global.asax

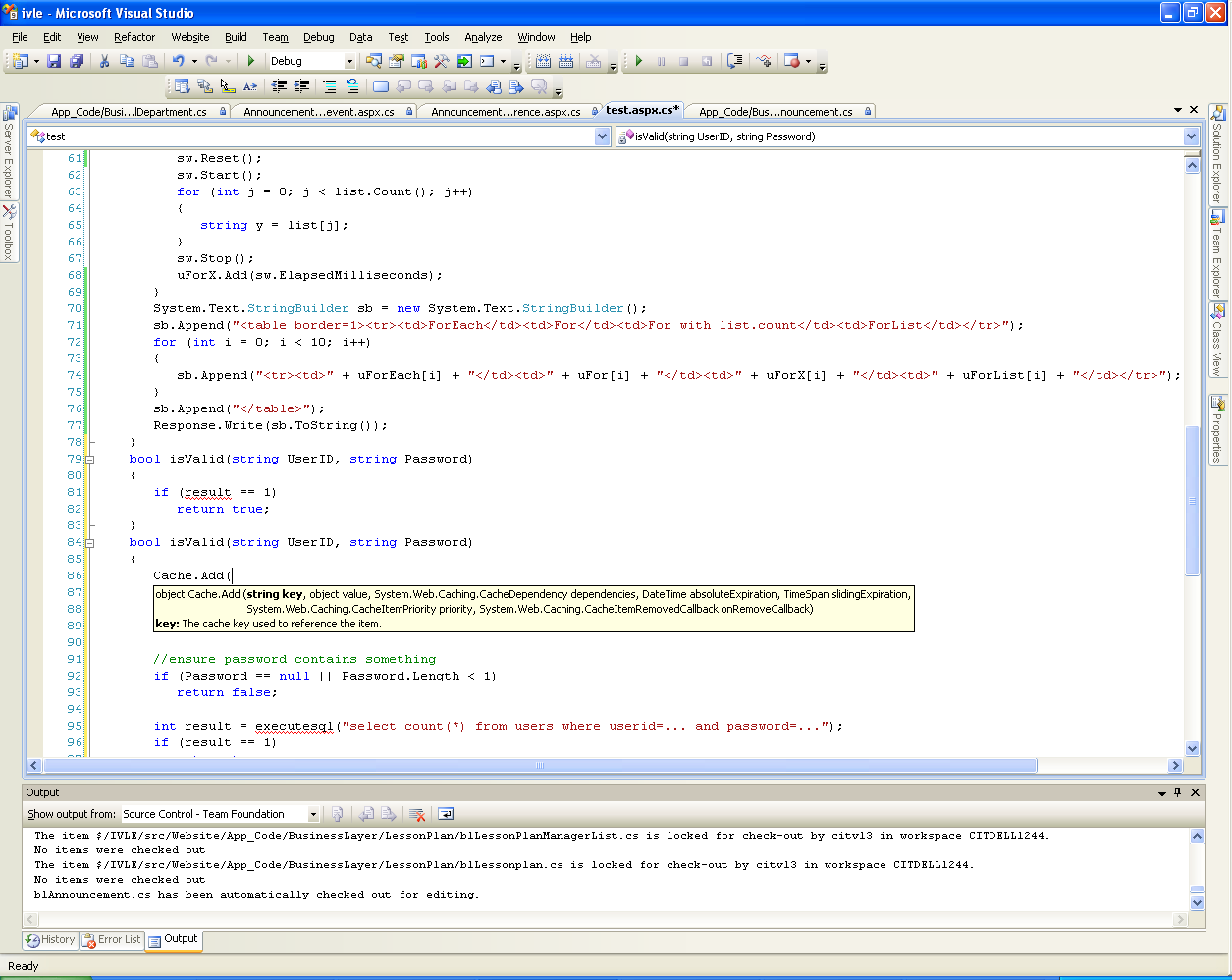
void Application\_Start(object sender, EventArgs e) {  
 Application["LastReload"] = DateTime.Now;   
}

inside your aspx pages

Response.Write((Application["LastReload"] as DateTime).ToString("dd MM yy");

**Caching**

If you have commonly used data which does not change often, you can consider storing it in cache to improve performance. Instead of calling the db to get a list of categories, call it once and store it into cache. The next time you need this list, get it from the cache instead. Remember though, you need to check whether the item returned from cache is null



object obj = Cache["key"];

if (obj == null)

{

//get the data from database

}

else

{

return (obj as List<string);

}

**Session State**

Use session state to store user specific properties. If you have no need for sessionstate, remember to turn it off in web.config using the following code

<sessionState mode="Off" />

As SessionState is the state which is usually kept the shortest, always check to ensure that the value returned is checked for null, this ensures that the NullValueException which can potentially be raised is not triggered.

object obj = Session["key"];

if (obj == null)

{

//get the data from database

}

else

{

return (obj as List<string);

}

**ViewState**

By default every page will have viewstate enabled. This will increase your page size along with serialization and deserialization costs. These costs can become very large especially when you have gridviews or datagrids in your page.

You should disable viewstate on the page if the page is an output page and does not do any further processing.

Additionally if the control does not handle any event, does not have any data bound values, or the data is constantly refreshed on postback, you can disable the viewstate for the control.

e.g you have a label which displays the userid with the value always refreshed on page load. In this case there is no need for viewstate.

<asp:Label ID="lbl" runat="server" />

and inside the page cs page

protected void Page\_Load(object sender, EventArgs e)

{

lbl.Text = User.Identity.Name;

**Extensions**

There are always times when we want to add a new function to an existing class which we do not have the code, and it is troublesome to create a new class just to add in this new function. What you can do is use extensions to add functions to existing classes.

e.g to add a Clear function to the StringBuilder class I can use the following

public static class StringBuilderExtensions

{

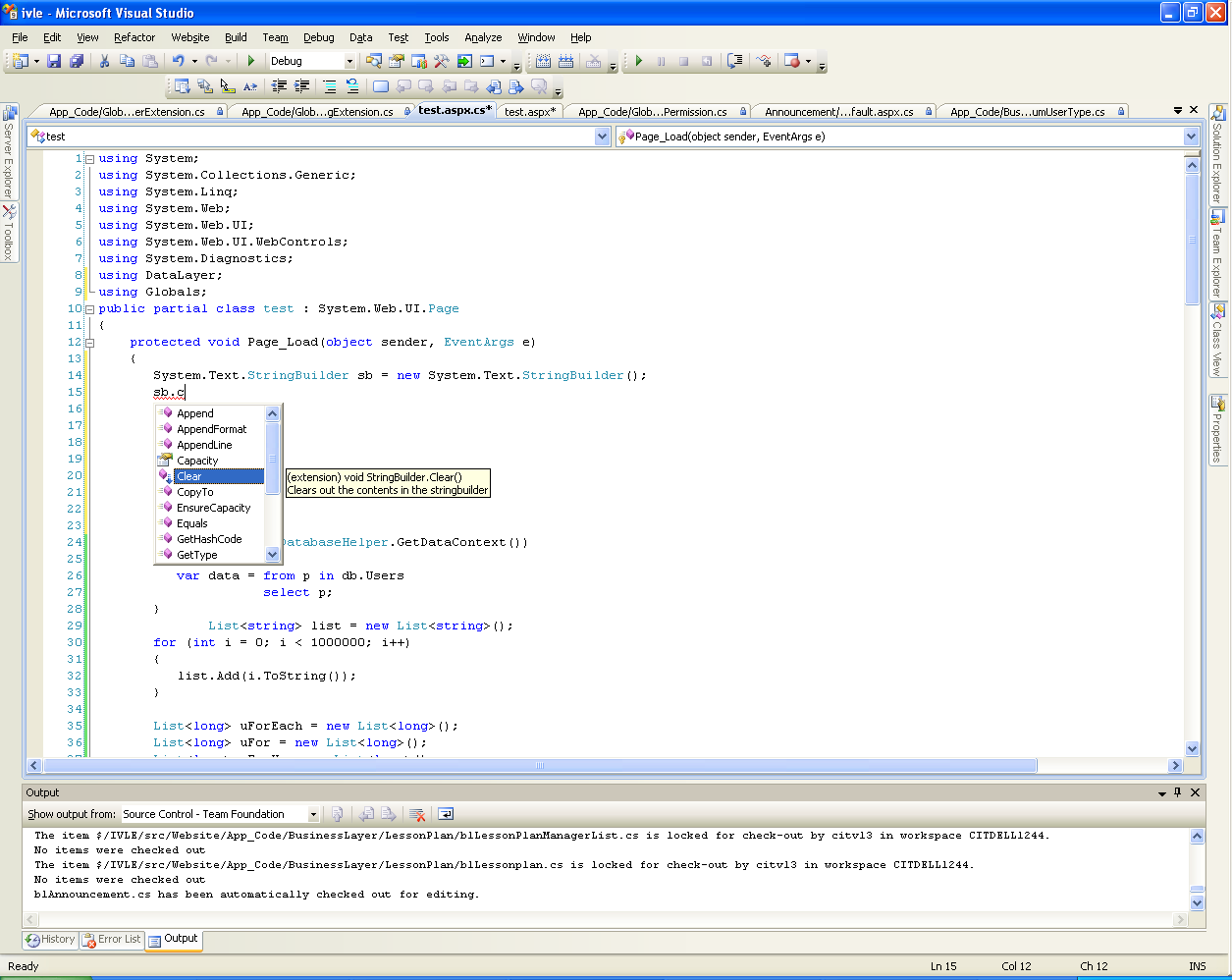
public static void Clear(this StringBuilder sb)

{

sb.Remove(0, sb.Length);

}  
}

And I can now call the function inside the page code behind



**Data Access**

**LINQ (Language INtegrated Query)**

LINQ allows for use of SQL like language inside code. For example

using (var db = DatabaseHelper.GetDataContext())

{

var users = from p in db.Users

select p;

}

It generates the DataContext using the database table mapping which you have to create. Using this DataContext, it can even autocomplete the table relations, or do various functions normally available only on the database server e.g distinct, group, average.

As everything in LINQ is based on objects, and the resultant SQL query being generated by ASP.NET, it reduces the surface areas for SQL injection attacks.

In fact, since it is all objects, you can easily refer to the fields of a table via the table’s properties. In addition, you can easily do inserts, updates and deletes via LINQ.

If however you use the same query multiple times, it will be better if you use compiled LINQ. What compiled LINQ does it that it compiles the LINQ query once, stores the resultant SQL and the delegate that is used to extract the data, so that it does not need to be compiled again a second time

So instead of using this every time you need to get a user record

from U in dc.Users

where U.UserID == userid

select U

You create a compiled LINQ reference and call the function instead

public static Func<DBDataContext, string, IQueryable<User>> CQ\_GetUserRecord = CompiledQuery.Compile<DBDataContext, string, IQueryable<User>>((dc, userid) =>

from U in dc.Users

where U.UserID == userid

select U

);