Performance Best Practices Checklist

Client Side:

- No inline JavaScript
- No Inline Styles
- Refer only the necessary include [*js*, *css*, etc.] files for the page
- Always remove code that you do not need
- Minimal comments, comment only what the code does and not how!
- Always Unbind before binding
- Use *JSON* instead of long cumbersome traditional response
- Use Firebug, Always, if you are not looking into Firebug at least five times a day you are really not justifying the tag of web developer! [there is a separate section in this document on what to check for in Firebug]
- Use *AJAX* as *AJAX* and Not JAX! Seriously, why use AJAX if we need to make every other call synchronous? Can be the good old code behind file! When you make *Asynch: false* make sure you have a reason to justify it.
- In jQuery Refer an element as directly as possible using the ID selector rather than using search/find in a container, yes that is right, because it translates directly to good old *getElementByID()*
- Avoid using pseudo and attribute selectors – they are the slowest!
- Chain as much as possible
- Make use of event bubbling – when many nodes need to call the same function
- *$(document).ready* is like HelloWorld of jQuery programming, jQuery doesn't start and end there, check the correct and appropriate place to call events and load things, load as late as possible and just in time
- String concatenation is Bad in client side too, use *join()* method to append lengthy strings and long running loops!
- Remember, *CSS* on top and *JS* on Bottom
- DRY! Don't Repeat Yourself!
- Loops are slow and bad in any language, use only when needed

- As much as possible Do NOT combine your JS logic and DOM loading, for example, it is Bad to write something like:

  ```
  for (var ct=0; ct <1000; ++ct)
  {
    $("#header").html($("#header").html() + ‘something from resultset’);
  }
  ```

  should be written as:

  ```
  var fullHeaderContent = $("#header").html();

  for (var ct=0; ct <1000; ++ct)
  {
    fullHeaderContent  += ‘something from resultset’;
   }
  $("#header").html(fullHeaderContent );


  //BAD
  for (var ctPatient=0; ctPatient<=rows.length; ++ctPatient)
  {
     $('#alertResult').append('<tr><td>'+rows[ctPatient]+'</td></tr>');
  }


  // AWESOME
  var domTree = '';
  for (var ctPatient=0; ctPatient<=rows.length; ++ctPatient)
  {
     domTree += '<tr><td>'+rows[ctPatient]+'</td></tr>';
  }
  $('#alertResult').append(domTree);
  ```

Firebug Usage:

- The best Javascript debugger out there!
- **Console** shows warning, errors, info
- Look for Yellow and Red – If you see them, they are bad, get rid of those warnings, errors right away!
- Do you see **404**? Bad, Very Bad, Please act on it
- Are there repeated calls? No use of "stressing" the point here, get rid of duplicate calls.
- "**Net**" tab – The Waterfall is your friend; Analyze calls, note ethe total time taken; if it is more than 2 seconds, we have an issue at hand! Click on every individual call, understand Waiting & Receiving times – Waiting times can be reduced with help of TRUE Asynch calls wherever possible
- Number of requests – can they be reduced? Are there repeated calls? Where is the most time spent?
- Categorize the bottlenecks using the waterfall charts
- console.profile, console.time and other Console APIs are very powerful, learn and use them http://getfirebug.com/wiki/index.php/Console_API
- Use the *YSlow* plugin to analyze project wide ratings are not dropping in your page, also see what can be done to improve every param at page level
- Advanced reading and great if you can use & check: https://code.google.com/p/leak-finder-for-javascript/

C#/Server side/EF Checklist:

- Use *Using* statements wherever possible
- Avoid Exceptions – aren't we sick of the *NULL* reference exception, be safe! Make Use of *TryParse* Etc! Also even when you use TryParse remember that it will handle just an *ArgumentException* and Not all types!
- After avoiding as much as possible, expect & handle Exceptions, even though there are some who argue that Exception Handling is costly, No, it is not an option not to have it at all!
- Exceptions cannot dictate how the application would behave, it is wrong to control application flow based on exceptions
- It is just not *try – catch*; let us not forget ***finally***! Also remember that just *try-finally* is right without a *Catch* if there is no need for a *Catch*
- It is better not to *throw*!
- When you catch an exception, creating an *object* and then throwing will alter *stacktrace*, not a best thing to have when we want to track and shoot down exceptions
- Never suppress exceptions – Empty *Catch* blocks; they serve NO purpose!
- Dispose objects, Yes, this is indeed a debatable point but let us be safer than sorry! It is always safer to call *Close* or *Dispose* for supported objects rather than not doing it; implement *IDisposable*!
- Acquire late, release early
- Declare as close as to usage as possible – don't do this when you have a really huge loop, better to declare outside!
- If we need to get 10 items, it is always preferable to get all 10 together rather than connecting/fetching 10 times!
- Reuse code as much as possible. DRY! Don't Repeat Yourself!
- Make sure to reuse common code wherever it is there, if it is not, it is never too late to start something, JUST DO IT!
- Avoid locks & synchronization as much as possible, use only when it is justified
- *Boxing* (and *unboxing*) is best left to stages and competitions, avoid as much as possible. This is where Generics help, before Generics we were left with only Objects, which would make the CLR to box unbox internally so many thousands of times in each application, each day!
- While using Generics, choose wisely *Stack<T>* for a LIFO; *Queue<T>* for a FIFO; *List<T>* for random access with zero based indexing; if no duplicates in the collection, always use

HashSet<T> - that is the fastest of the lot; Always Prefer *Dictionary<TKey,List<TValue>>* over List *<KeyValuePair<TKey,TValue>>* whenever possible for representing a one-to- many relationship between two entities. Lookup speed will be much faster and client code will be less clumsy.

- Prefer LINQ Standard Query Operator *Where()* over *Contains(), Exists(),* or *Find()* methods to check whether a value is present in a List<T> instance.
- Use the *OrderBy()* or the *OrderByDescending()* method to sort any *IEnumerable<T>*
- Avoid using of AsEnumerable() as it processes on memory rather than on DB.
- Avoid creating unnecessary objects
- Prefer Query Syntax over looping! Looping is heavy in any programming language!
- Firing queries inside loop is bad and should be avoided, as much as possible get all necessary data in the outer query
- Be aware that comparing unconstrained type parameter values against *Null* is extremely slow for *nullable* value types
- Use String Builder over Concat when you need to append more than 6 or 7 string instances
- Always return only what is necessary
- With fields of type derived from *MarshalByRefObject*, the performance is particularly bad because the CLR really ends up calling methods to perform the field access.
- We all know that reflection is bad, so please use it only it is absolutely needed like creation of all new dynamic runtime forms Etc.
- *DataBinder.Eval* is to be avoided – it uses reflection internally
- *Virtual* methods are great but come with a performance overload, so *seal* your classes whenever possible and when you don't mean to inherit them anymore!
- Set Sessionstate to read only when you do not write anything on a services, controllers, Etc.
- Foreach on a list is proven to have performance issues, avoid if possible
- Know what to optimize, do not shoot in the dark!
- Use the AjaxRender Helper class [which is a rendering engine] wherever applicable to allow Asynch loading
- Stay out of threads unless you are a Master in it and understand then 100%
- Read about Synchronous and Asynchronous pipelines in MVC and make sure to use the appropriate one at appropriate place
- **** ALWAYS Use Glimpse or Mini Profiler – They just have many surprises for you when not taken care of turn into nasty surprises!!

- Explicitly tell EF the type of the expected result wherever possible to avoid
- When you want to retrieve a single entity using entity key can be more efficient than using WHERE clause [use of GetObjectByKey() method]
- To efficiently retrieve entities that will only display and not update, use the MergeOption.NoTracking merge option
- Avoid "*let*" http://msdn.microsoft.com/en-us/library/bb383976.aspx in your queries as much as possible
- Consider using Compiled Queries – CompileQuery.Compile() [note that this decreases readability]
- ALWAYS see the actual SQL generated – you will have many surprises here, you can use LINQPad or MiniProfiler-EF or simply the trace form SQL – this is very important, this will tell you what exactly happens with your LINQ queires
- I would personally advise SPs for anything complex [though opinion is divided here]
- Limit the scope of the ObjectContext
- Once again, get ONLY the records that you need!
- Try to use columns in WHERE clause in a logical order where the most likely indexed columns are first, this is debatable if SQL Server really cares for the Order but better be safe than sorry
- SQL doesn't care about CASE so do not bother about using ToLower() Etc in your LINQtoSQL
- "*for*" is the fastest way of iterating over a collection, *foreach* is a little slower, and LINQ queries are slowest
- Disable *Viewstate* when not used and when you rebind page every time

SQL Checklist:

- Do NOT use Select * - it is Bad, Bad, Bad!
- Do NOT use IN Queries where JOIN will do the same
- Do NOT use NOT IN, LEFT OUTER is there for some reason ;)
- Do NOT use the old fashioned way of JOINs [the entire table list in FROM clause]
- This is not really a performance point but do NOT use Left table's conditions in WHERE – it will make it an INNER JOIN!
- Column order in WHERE clause - Give priority to columns that are most likely to be indexed
- I have seen Queries which are Super safe! Multiple conditions in WHERE clause even after selecting with the Primary Key!! Seriously?
- In PSQL, make sure to declare the variables which are the same as of the column
- When 2 tables have FKs, the type should be the same [of course] else will result in Casting and overhead
- Do not use Cursors, Loops as much as possible and stay away from temp tables, consider views in their place!
- Make cursors Read Only Fast forward when you do not manipulate anything in them
- Make use newer datatypes like Date, Time, DateTime2 and so on! Make sure to use the appropriate datatype a BIGINT column in place of a TINYINT column in table with Millions of rows will mean GBs of wasted space – resulting in data being scattered
- Index, Index, Index
- Right Index!
- Do NOT index when you should not – sample a column with very little variance of data [something like row_status] with millions of rows of data will affect more than helping [it is simple isn't it, Indexes are B Tree structures and if all possible values are just 0, 1, 2 we are looking at a really small B-Tree ;)]
- Learn SQL tracing, it is exciting, superb and tells you man a stories
- Look at Actual Execution Plans - Do You see a Index Seek – great, Index Scan – alright; is there anything else like tablescan Etc? We have a problem! See where is the most time spent, look at suggested indexes – look at reads, are there too many?
- Consider using Include indexes
- ORDER BY Clauses need Indexing too and it makes sense to Index also in DESC order for things like Dates Etc which are mostly shown in Desc order!

- Use Narrow Indexes when possible

- As with anything Indexes again can make or break, make sure to test thoroughly with reasonable amount of data, execution plans differ when there is a huge difference in amount of data
- Leave hints alone, let Optimizer decide the best plan
- When you want to process things like what is the status of something for each person or car or whatever, it is ALWAYS best to do as a single Query for everything together than firing 1000s of queries for each person/car – Remember to work in sets rather than with each individuals row!
- Avoid using functions in predicates
- Avoid using wildcard (%) at the beginning of a predicate
- DISTINCT and UNION should be used only if it is necessary.
- Use constants and literals if the values will not change (for static queries)
- Make numeric and date data types match