Good Performance: Three Developers' Behaviors that Prevent It!

Introduction

Performance is regarded as one of the most valuable non-functional requirements of an application. If you are reading this, you are probably using an application like a browser or document reader, and understand how important performance is. In this article, I will talk about applications' good performance and three developers' behaviors that prevent it.

Behavior #1: Lack of understanding of the development technologies

It doesn't matter whether you are someone who just graduated from school or have years of experience; when you have to develop something, you will probably look for something that was already developed. Hopefully in the same programming language.

This is not a bad thing. In fact, it often speeds up development. But, on the other hand, it also might prevent you from learning something. Because only rarely does this approach involve taking the time to inspect the code and understand not only the algorithm but also the inner workings of each line of code.

That is one example of us, as developers, falling into behavior number one. But there are other ways too. For example, when I was younger and just starting my journey in software development, my boss at the time was my role model, and whatever he did was the best someone could do. Whenever I had to do something, I looked at

how he did it and replicated it as closely as possible. Many times, I did not understand why his approach worked, but who cares, right? It worked!

There is a kind of developer that I call a "4x4." He or she is someone, who when asked to do something works as hard as possible to complete it. They usually look for building blocks, or pieces of things already done, puts them all together, and "voilà!" The thing is done! Rarely does this kind of developer spend any time understanding all the pieces he or she found and don't consider or investigate scalability, maintainability, or performance.

There is one more situation that leads to developers not understanding how things actually work: never running into problems! When you use a technology for the first time and you run into problems, you dig into the details of the technology, and you end up understanding how it works.

At this point, let's look at some examples that will help us understand the difference between understanding the technology and simply using it. Since I am, for the most part, a .NET* web developer, I will focus on that.

JavaScript* and the Document Object Module (DOM)

Let's look at the code snippet below. Pretty plain. The code just updates the style of an element in the DOM. The problem (which is less of a problem with modern browsers but included to illustrate the point), is that it is traversing the DOM tree three times. If this code is repeated and the document is large and complex, there will be a performance hit in the application.

Code #1

```
document.getElementById("myElement").style.color = "#ccc";
document.getElementById("myElement").style.backgroundColor = "#fcc";
document.getElementById("myElement").style.fontWeight = "bold";
```

Fixing such a problem is easy. Look at the following code snipped. There is a direct reference hold in the variable *myField* prior to working on the object. This new code is less verbose, quicker to read and understand, and has better performance since there is only one access to the DOM tree.

Code #2

```
var myField = document.getElementById("myElement");

myField.style.color = "#ccc";
myField.style.backgroundColor = "#fcc";
myField.style.fontWeight = "bold";
```

Let's look at another example. This example was taken from: http://code.tutsplus.com/tutorials/10-ways-to-instantly-increase-your-jquery-performance--net-5551

In the following figure, there are two equivalent code snippets. Each code creates a thousand list item *li* elements. The code on the right adds an *id* attribute to each *li* element, whereas the code on the left adds a *class* attribute to each *li* element.

Classes	Ids

`var list = $('#list');`
`var items = '<ul>';`
 	
`for (i = 0; i < 1000; i++) {`
`  items += '<li class="item' + i + '">item</li>';`
`}`
 	
`items += '</ul>';`
`list.html(items);`
	`var list = $('#list');`
	`var items = '<ul>';`
	
	`for (i = 0; i < 1000; i++) {`
	`  items += '<li id="item' + i + '">item</li>';`
	`}`
	
	`items += '</ul>';`
	`list.html(items);`

`console.time('class');`
 	
`for (i = 0; i < 1000; i++) {`
`  var s = $('.item' + i);`
`}`
 	
`console.timeEnd('class');`
	`console.time('id');`
	
	`for (i = 0; i < 1000; i++) {`
	`  var s = $('#item' + i);`
	`}`
	
	`console.timeEnd('id');`

As you can see, the second part of each code snippet simply accesses each of the thousand *li* elements that was created. In my benchmarking in Internet Explorer* 10 and Chrome* 48, the average time taken was 57 ms for the code on the left and 9 ms for the code on the right—significantly less. The difference is huge in this case when just accessing the elements in one way or the other.

This example has to be taken very carefully! There are so many additional things to understand that might make this example look wrong, like the order in which the selectors are evaluated, which is from right to left. If you are using jQuery*, read about the DOM context as well. For general CSS Selectors' performance concepts, see the following article: https://smacss.com/book/selectors

Let's provide a final example in JavaScript code. This example is more related to memory but will help you understand how things really work. High memory consumption in browsers will cause a performance problem as well.

The next image shows two different ways of creating an object with two properties and one method. On the left, the class's constructor

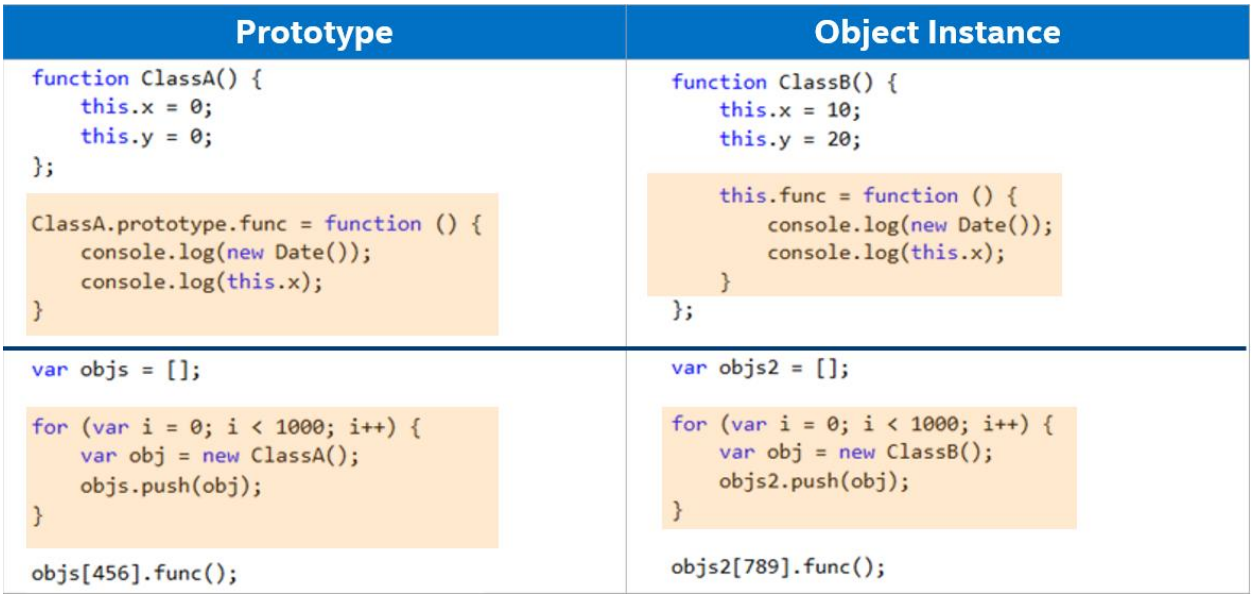
adds the two properties to the object and the additional method is added through the class's prototype. On the right, the constructor adds the properties and the method at once.

After the objects are created, a thousand objects are created using both techniques. If you compare the memory used by the objects you'll see differences of memory usage in the Shallow Size and Retained Size for both approaches in Chrome. The prototype approach uses about 20 percent less memory (20 Kbytes versus 24 Kbytes) in the Shallow Size and references up to 66 percent less in Retained Memory (20 Kbytes versus 60 Kbytes).

For a better understanding of how Shallow Size and Retained Size memory work, see:

https://developers.google.com/web/tools/chrome-devtools/profile/memory-problems/memory-101?hl=en

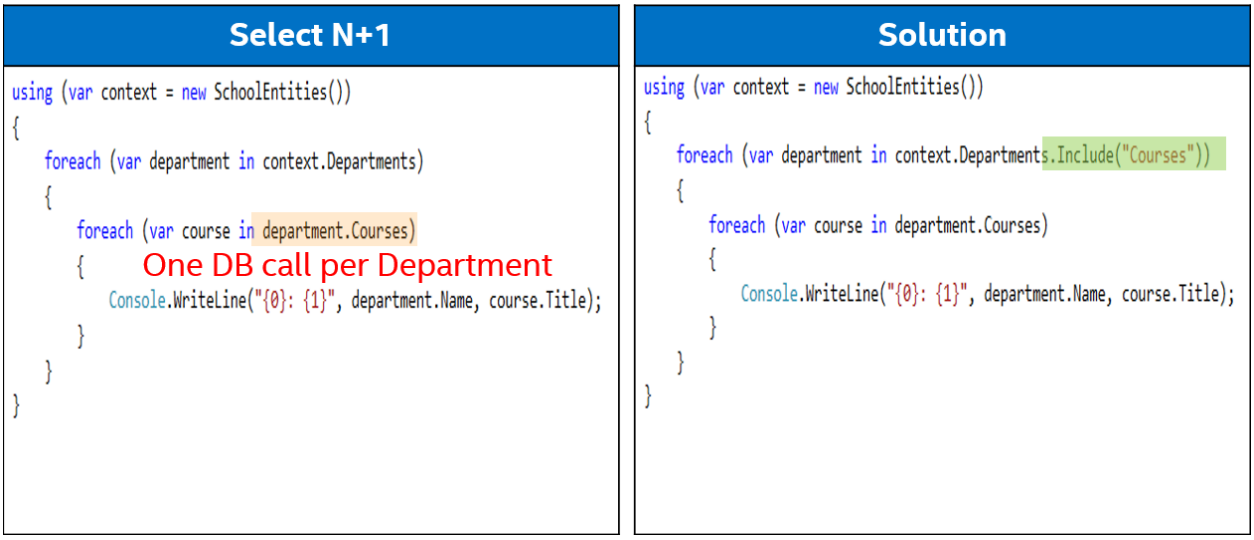
You can create objects by knowing how to use the technology. But understanding how the technology actually works gives you tools to improve the application in areas like memory management and performance.



LINQ

When I was preparing my conference presentation on this topic, I wanted to provide an example with server-side code. I decided to use LINQ*, since LINQ has become a first-hand tool in the .NET world for new development and is one of the most promising areas to look for performance improvements.

Consider this common scenario. In the following image there are two functionally equivalent sections of code. The purpose of the code is to list all departments and all courses for each department in a school. In the code titled Select N+1, we list all the departments and for each department list its courses. This means that if there are 100 departments, we will make 1+100 calls to the database.



There are many ways to solve this. One simple approach is shown in the code on the right side of the image. By using the *Include* method (in this case I am using a hardcoded string for ease in understanding) there will be one single database call in which all the departments and its courses will be brought at once. In this case, when the second *foreach* loop is executed, all the Courses collections for each department will already be in memory.

Improvements in performance on the order of hundreds of times faster are possible simply by avoiding the Select N+1 problem.

Let’s consider a less obvious example.

In the image below, there is only one difference between the two code snippets: the data type of the target list in the second line. You might ask, what difference does the target type make? When you understand how the technology works, you will realize that the target data type actually defines the exact moment when the query is executed against the database. That, in turn, defines when the filters of each query is applied.

In the case of the Code #1 sample where an *IEnumerable* is expected, the query is executed right before *Take<Employee>(10)* is executed. This means that if there are 1,000 employees, all of them will be retrieved from the database and then only 10 will be taken.

Code #1

```
MyDataContext dc = new MyDataContext();
IEnumerable<Employee> list = dc.Employees.Where(p => p.Name.StartsWith("S"));
list = list.Take<Employee>(10);
```

```
SELECT [t0].[EmpID], [t0].[EmpName], [t0].[Salary] FROM [Employee] AS [t0]
WHERE [t0].[EmpName] LIKE @p0
```

Code #2

```
MyDataContext dc = new MyDataContext();
IQueryable<Employee> list = dc.Employees.Where(p => p.Name.StartsWith("S"));
list = list.Take<Employee>(10);
```

```
SELECT TOP 10 [t0].[EmpID], [t0].[EmpName], [t0].[Salary] FROM [Employee] AS [t0]
WHERE [t0].[EmpName] LIKE @p0
```

In the case of the Code #2 sample, the query is executed until *Take<Employee>(10)* is executed. That is, only 10 records are retrieved from the database.

The following article has an in-depth explanation of the differences in using multiple types of collections.

http://www.codeproject.com/Articles/832189/List-vs-IEnumerable-vs-IQueryable-vs-ICollection-v

SQL Server*

In SQL, there are many concepts to understand in order to get the best performance possible out of your database. SQL Server is complex because it requires an understanding of how the data is

being used, and what tables are queried the most and by which fields.

Nevertheless, you can still apply some general concepts to improve performance, such as:

- Clustered versus non-clustered indexes
- Properly ordered JOINs
- Understanding when to use *#temp* tables and variable tables
- Use of views versus indexed views
- Use of pre-compiled statements

For the sake of brevity, I won't provide a specific use case, but these are the types of concepts that you can use, understand, and make the most of.

Mindset Change

So, what are the mindset changes we, as developers, must have in order to avoid behavior #1?

- Stop thinking "I am a front-end or back-end developer!" You probably are an engineer and you may become an expert in one area, but don't use that as a shield to avoid learning more about other areas.
- Stop thinking "Let's let the expert do it because he's faster!" In the current world where agile is all over the place, we must be fungible resources, and we must learn about the areas we are weak in.
- Stop telling yourself "I don't get it!" Of course! If it was easy then we all would be experts! Spend your time reading, asking, and understanding. It's not easy, but it pays off by itself.
- Stop saying "I don't have time!" OK, I get this one. It does happen. But once an Intel fellow told me "when you are

> passionate about something, your bandwidth is infinite." And here I am, writing this article at 12:00 a.m. on a Saturday!

Behavior #2: Bias on specific technologies

I have developed in .NET since version 1.0. I knew every single little detail of how Web Forms worked as well as a lot of the .NET client-side libraries (I customized some of them). When I saw that Model View Controller (MVC) was coming out, I was reluctant to use it because "we didn't need it."

I won't continue with the list of things that I didn't like at the beginning but now use extensively. But this makes my point of people's bias against using specific technologies, preventing themselves from getting better performance.

One of the discussions I often hear is either about LINQ-to-Entities in Entity Framework, or about SQL Stored Procedures when querying data. People are so used to one or the other that they try to continue using them.

Another aspect that makes people biased toward a particular technology is whether they are open source lovers or haters. This makes people not think about what is best for their current situation but rather what best aligns to their philosophy.

Sometimes external factors (for instance, deadlines) push us to make decisions. In order to choose the best technology for our applications, we require time to read, play, compare, and conclude. When we start developing a new product or version of an existing product, it's not uncommon that we are already late. Two ways

come to mind on how to solve this situation: stand up and ask for that time or work extra hours to educate ourselves.

Mindset Change

So what are the mindset changes we, as developers, must have in order to avoid behavior #2:

- Stop saying "This has always worked," "This is what we have always used," and so on. We need to identify and use other options, especially if there is data that supports those options.
- Stop fixing the solution! There are times when people want to use a specific technology that doesn't provide the expected results. Then they spend hours and hours trying to tweak that technology. What they are doing in this case if "fixing the solution" instead of focusing on the problem and maybe finding a quicker, more elegant solution somewhere else.
- "I don't have time!" Of course, we don't have time to learn or try new stuff. Again, I get this one.

Behavior #3: Not understanding the application's infrastructure

After we have put a lot of effort into creating the best application, it is time to deploy it! We tested everything. Everything worked beautifully in our machines. All the 10 testers were so happy with it and its performance. So, what could go wrong after all?

Well, everything could go wrong!

Did you ask yourself any of the following questions?

- Was the application expected to work in a load-balanced environment?

- Is the application going to be hosted in the cloud with many instances of it?
- How many other applications are running in my target production machine?
- What else is running on that server? SQL Server? Reporting Services? Some SharePoint* extensions?
- Where are my end users located? Are they all over the world?
- How many final users will my application have in the next five years?

I understand that not all of these questions refer to the infrastructure but bear with me here. More often than not, the final conditions under which our application will run are not the same as our staging servers.

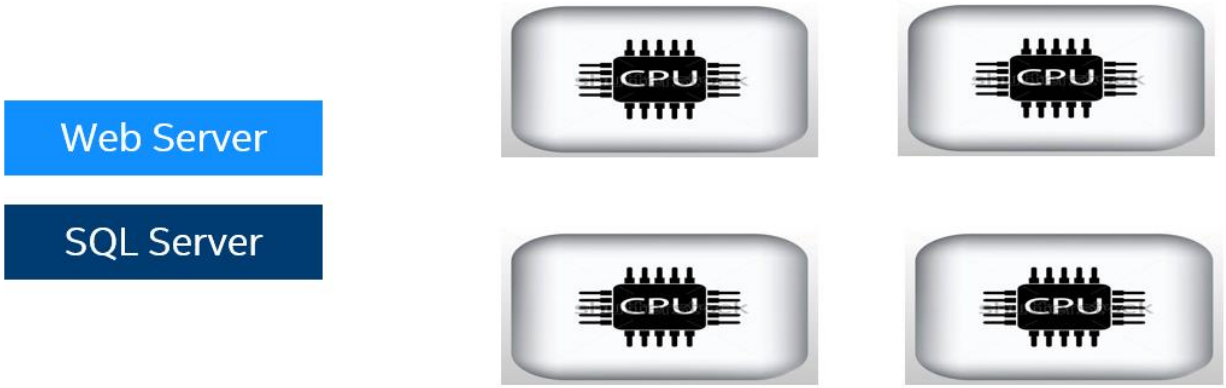
Let's pick come of the possible situations that could affect the performance in our application. We will start with users around the world. Maybe our application is very fast and we hear no complaints from our customers in America, but our customers in Malaysia don't have the same speedy experience.

There are many options to solve this situation. For one, we could use Content Delivery Networks (CDNs) where we can place static

files, then loading pages would be faster from different locations. The following image shows what I am talking about.

Picking another potential situation, let's consider applications running on servers having SQL Server and Web Server running together. In this case we have two CPU-intensive servers on the same machine. So, how can we solve this? Still assuming you are running a .NET application in an Internet Information Services (IIS) Server, we could take advantage of CPU affinity. CPU affinity ties one process to one or more specific cores in the machine.

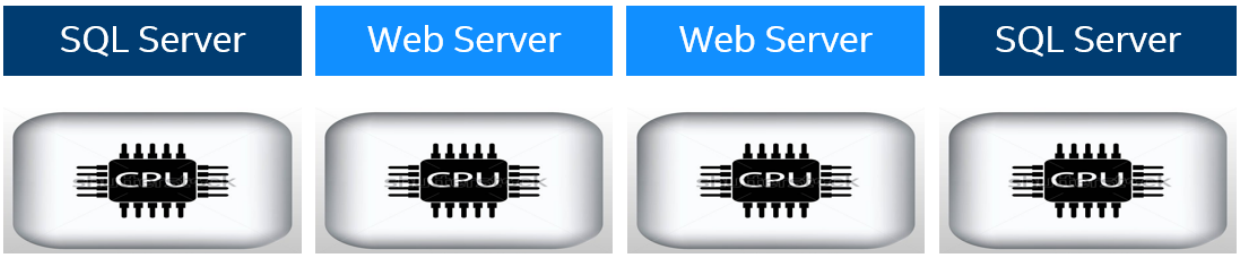
For example, let's say that we have SQL Server and Web Server (IIS) in a machine with four CPUs.



If we leave it to the operating system to determine what CPU uses

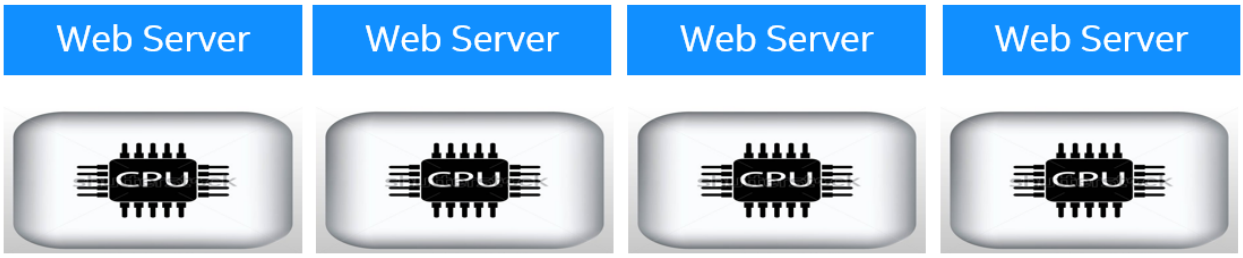
the IIS or the SQL Server, there could be various setups. We can have two CPUs assigned to each server.

Good !



Or we could have all processors assigned to only one server!

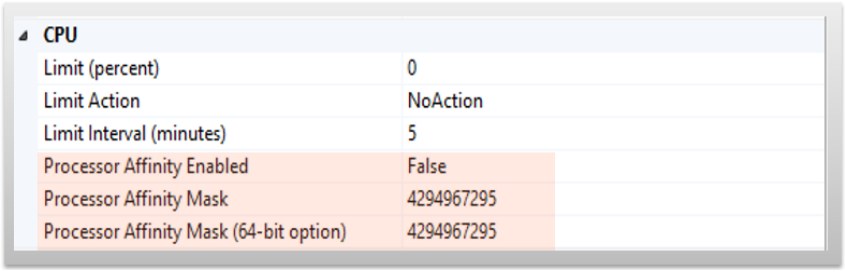
Bad for SQL Server!



In this case, we could fall into deadlock because the IIS might be attending too many requests that took all four processors, and probably some of them will require access to SQL, which, of course, won't happen. This is admittedly an unlikely scenario, but it illustrates my point.

There is one additional situation: one process will not run on the same CPUs all the time. There will be too much context switching. This context switching will cause performance degradation in the server and then in the applications running on that server.

One way to minimize this is by using processor affinity for IIS and SQL. In this way, we can determine how many processors we need for the SQL Server and for the IIS. This is done by changing the Processor Affinity settings in the CPU category in the IIS and the "affinity mask" in the SQL server database. Both cases are shown in the following images.




```
sp_configure 'show advanced options', 1;
RECONFIGURE;
GO
sp_configure 'affinity mask', 38;
RECONFIGURE;
GO
```

I could continue with other options at the infrastructure level to improve applications' performance, like the use of Web Gardens and Web Farms.

Mindset Change

What are the mindset changes we, as developers, must have in order to avoid behavior #3?

- Stop thinking "That is not my job!" We, as engineers, must broaden our knowledge as much as possible in order to provide the best integral solution to our customers.

- “I don’t have time!” Of course, we never have time. This is the commonality in the mindset change. Making time is what differentiate a professional that succeeds, exceeds, or outstands!

Don’t feel guilty!

But, do not feel guilty! It is not all on you! Really, we don’t have time! We have family, we have hobbies, and we have to rest!

The important thing here is to realize that sometimes there is more to performance than just writing good code. We all have shown and will show some or all of these behaviors during our lifetime.

Let me give you some tips to avoid these behaviors.

1. Make time. When asked for estimates for your projects, make sure you estimate for researching, testing, concluding, and making decisions.
2. Try to create along the way a personal test application. This application will avoid you having to try stuff in the application under development. This is a mistake we all make at some point.
3. Look for people that already know and do pair-programming. Work with your infrastructure person when he or she is deploying the application. This is time well spent.
4. Stack Overflow is evil!!! Actually, I do help there and a big percentage of my problems are already solved there. But, if you use it for “copy and paste” answers, you will end up with incomplete solutions.
5. Stop being the front-end person. Stop being the back-end person too. Become a subject matter expert if you will, but

make sure you can hold a smart discussion when talking about the areas where you are not an expert.

6. Help out! This is probably the best way to learn. When you spend time helping people with their problems, you are in the long run saving yourself time by not encountering the same or similar situations.

See Also

Funny and interesting blog from Scott Hanselman about being a googler or a developer.
http://www.hanselman.com/blog/AmIReallyADeveloperOrJustAGoodGoogler.aspx

More about objects and prototypes:
http://thecodeship.com/web-development/methods-within-constructor-vs-prototype-in-javascript/