Annals of DAAAM for 2012 & Proceedings of the 23rd International DAAAM Symposium, Volume 23, No.1, ISSN 2304-1382
ISBN 978-3-901509-91-9, CDROM version, Ed. B. Katalinic, Published by DAAAM International, Vienna, Austria, EU, 2012
Make Harmony between Technology and Nature, and Your Mind will Fly Free as a Bird
Annals & Proceedings of DAAAM International 2012

EMPIRICAL COMPARISON OF DIFFERENT VIEW STATE APPROACHES ON PERFORMANCE OF ASP.NET WEB APPLICATIONS

DJAMBIC, G[oran]; KUCAK, D[anijel] & FULANOVIC, B[ojan]

Abstract: *Web forms are still common choicefor development of ASP.NET applications, despite the fact that ASP.NET MVC is becoming increasingly popular. One of the main paradigms of Web forms are postback and ViewState. A situation when Web form issues POST request back to itself is called a postback. ViewStatemechanism was introduced to bridge the gap between postbacks. It provides an illusion of statefulness between two requests on the same Web form, as if the state of Web form hasn't changed since the last request was finished processing. In order to utilize ViewState no configuration is needed and it is common practice to leave default ViewState settings. This paper shows how ViewState size affects performance of Web form. It shows that ViewState size can easily grow large and that default settings can affect performance by enlarging both responses and requests. Two alternatives are analyzed: sending compressed ViewState to client and leaving ViewState on server.We show that best choice is to leaveViewState on server. Since that method is not always applicable, we show that the second best method is compression. It is applicable in almost all situations and will have good impact on performance.*

Keywords:*ASP.NET, Web forms, performance, ViewState, compression, Session*

1. INTRODUCTION

ASP.NET is a technology for building websites that has been around since 2001 andat the time of writing this paper (October2012)current version is 4. According to [1], 21% of top 1 million websites ([2]) use ASP.NET as their server-side platform, which should make the topic of this paper reasonably interesting.

From its first version, ASP.NET came with a programming paradigm called Web forms. Its basic idea was to encapsulate and hide the complexity of the HTTP request-response model and enable the programmer to build websites using a drag-n-drop approach and familiar event-driven model, just like when building desktop applications ([3]). This was made possible by use of Web server controls; framework-supplied or user-made controls that were designed to retain its state across multiple request-responses, thus creating an illusion of desktop behavior.

From the ASP.NET version 3.5, another programming framework was made available to programmers. It was designed around model-view-controller pattern and was aptly named MVC. Each platform has its strength and weaknesses ([4], [5]) and choosing one over another is driven by many factors. One of the main strengths of Web forms is considered to be its age of 10+ years, which means that there exists a lot of knowledge about it and a large community of adopters.

One of the main drawbacks of Web forms is considered to be ViewState. As said in [4], a ViewState is a mechanism that allows the last known state of each Web formto be stored somewhere and then restored for the next user request, in order to bridge the HTTP gap and to achieve statefulness. It is considered to be a drawback mainly because, by default, ViewState is sent to the client in the HTTP response and then again sent back to the server in the following the HTTP request, increasing the size of both requests and responses and increasing the processing time on the server.

This work will measure which approach to the ViewStatemanagement is the best for the performance of ASP.NET applications. To authors' knowledge, there are no similar attempts to quantify the performance loss because of the ViewState. There are other papers that are concerned with ASP.NET performance (such as [7] and [8]), but not particularly with the ViewState.

We will completely disregard the MVC, since there's no concept of a ViewState in it, and concentrate on ViewState performance impact on Web forms. As [6] indicates, almost a third of all considered web sites now use ASP.NET version 4. It can be seen as very high, because version 4 became available in 2010. Almost two thirds of considered web sites use older version 2, and that is one of the limitations of this work. Another limitation can be found in the fact that only in-memory Session storage is used, and that impact of different storage schemes is not analyzed.

We will consider three approaches to managing the ViewState and will measure their performance impacts: default approach, compressed ViewState approach and leaving the ViewState on a server approach. There are anumber of aspects that won't be considered in this work and that leave space for future work. We will consider only one way of compression, GZip, in which way the impact of other compression types can be examined. We will consider typical real-life data consisting mostly of integers and strings, so impact on compression of other data types can be examined, particularly if data gets encrypted also. We will consider storing ViewState on the server in in-memory Cache object, in which way the influence of other types of storages (e.g. disk, database, out-of-process) can be investigated.Since there are significant drawbacks of using in-memory Cache object as ViewState storage, some alternative scheme might be presented and evaluated against this one. Finally, one way to improve the ViewState performance is not to use a ViewState at all. It might be interesting to see a comparison of implementing standard functionalities without the ViewState and seeing its impact on the performance.

2. BACKGROUND

This section will explain in more detail how the ViewState mechanism works. In order to understandthe ViewState,we must firstly understand what is a postback, what are Web server controls and how do they work.

A postback is a process when the Web page issues the HTTP POST request back to itself in order to interact with its user. That process is the cornerstone of Web pages and ASP.NET applications depend heavily on it. Here is an example of postback: user issues a HTTP GET request to the Web form and receives an HTTP response containing an HTML document with two SELECT tags in its BODY, one filled with countries and one empty for cities. User chooses acountry and then ASP.NET generated JavaScript comes to scene. It causes a HTTP POST request to the same Web form – that is called a postback. The Web form receives the request, sees that the user has chosen a country, fills SELECT tag with cities for that country and sends response to client. Some complex Web forms can implement their functionality with use of many postbacks. All complexity for achieving postbacks is hidden in ASP.NET framework and a programmer only has to say which Web server controls require postback for which functionality.

We move onto the next important concept. On a Web Form we can find pure HTML, CSS and JavaScript elements, just like on any ordinary HTML page. But, on a Web Form we can also find Web server controls. Web server control is a class that inherits built-in class called System.Web.UI.Controland can be written in any .NET programming language. Its purpose is to emit a HTML (with CSS and JavaScript) to a client, while providing a strong programming model on the server based on events. On postbacks, ASP.NET framework raises control's events based on user's actions on a client. Very important point here is that for each request-response pair, a new instance of Web page and a new instance of every Web server control is created by ASP.NET environment. It is used to serve only the current request and after the response is sent, garbage control collects all instances. So when next postback request comes in, a new Web page instance and a new Web server control instances are created again, and there is the reason for ViewState existence: those new instances do not know anything about the states of previous instances, but for programmers, it would be very nice if they would know about them.

In order to bridge the mentioned HTTP gap and provide an illusion of Web form and its Web server controls maintaining their states across request-responses, a ViewState mechanism is introduced, as shown in the following, somewhat simplified, algorithm:

1. The first request (HTTP GET) arrives on the server and ASP.NET framework starts processing it. It creates a new instance of the Web form and a new instance for each child Web server control on the form. It alsoinitializes their properties to values usually defined in ASPX file.
2. Change tracking mechanism is activated – every change to every property is recorded.
3. The Web page instance goes through the rest of its life cycle and its different methods get invoked by the ASP.NET runtime ([4]).
4. At the near end of a life cycle, the Web form instance picks all recordedchanges since the activation of change tracking mechanism and writes those changes to a simple (although possibly very large) BASE64 encoded string. Default behavior is to send that string to a client as a value of an HTML HIDDEN INPUT tag named __VIEWSTATE.
5. At this point, all instances are garbage collected, response is sent to a client and now the user can interact with browser rendering the page.
6. The user makes an action that requires a postback and a new HTTP POST request comes to the server.
7. At the very early stage of processing this new request (just after the change tracking is activated), the values from the ViewState string are extracted and applied to new instances – in this way, we actually get old values copied to new instances, thus providing the illusion of continuity.

Previous algorithm is repeated until user browses away from the Web form.

There is another way of using a ViewState that is not related to Web server controls: programmers can manually put objects into the ViewState. Those objects must be serializable (because ViewState is a string) and must be put into the ViewState after the change tracking mechanism is activated and before the ViewState string is created for sending to a client. Those manually inserted objects travel with Web server controls properties and are also available on the subsequent requests.

3. METHODOLOGY

In this work we will use only manually inserted objects and not Web server control properties. Reasons for that are (1) the rest ofthe HTML is not changed, as opposed when using Web server control properties, and (2) we can explicitly control the ViewState size by inserting more or less objects. This will not make any difference because the distinction between Web server control properties and manually inserted objects disappears once they're serialized into the ViewState.

All tests have been run on the same machine and there were no changes in the hardware configuration during tests; the only thing that has been changing in tests was application parameters.Exact hardware specifications are not relevant since we are only interested in finding which of three approaches yields betterperformance in comparison to others, not their absolute values.

The performance parameter we are interested in is the time between the browser issues a request and until it receives the response. We consider that the performance is better when that time is shorter. We use the following equation to express that time:

$$t = t_{req} + t_{init} + t_{load} + t_{process} + t_{save} + t_{resp} \qquad (1)$$

where:

- t_{req} is the time from initiating a request from the browser until the request is received by the web server.
- t_{init} is the time from the moment when a request is received by the web server until the ASP.NET runtime has created all instances and had applied values from ASPX to them.
- t_{load} is the time necessary for extractingViewStatevalues from storageand applying them to appropriate properties or deserialize them into objects programmers put in.
- $t_{process}$ is the time that takes the ASP.NET to finish processing all methods in its life cycle and ends right before the ViewState is being made.
- t_{save} is the time that takes to create a ViewState.
- t_{resp} is the time from start of sending the response to the moment when a browser receives the response.

By minimizing *t* we optimize performance. $t_{init}$and $t_{process}$cannot be influenced, so we look at the performance impact of remaining times in minimizing *t*.

It is clear from their definition that t_{req} and t_{req} have a linear dependency on the size of request and response; the less bytes to transmit, the faster the transmission will be completed. We say that lesser is the ViewState sent to the client, better the performance will be, providing that t_{load} and t_{save}remain unchanged or changed just slightly so the entire *t* will remain minimized.

In our tests we put objects into the ViewState. Those objects are instances of a class consisting of 20 properties: 7 integers, 11 strings, 1 datetime and 1 GUID. That is important information in order to interpret compression results; all used data types are considered good for compression. Objects were filled from production database so the data distribution should correlate with the real-life scenarios.

In all tests we use the ViewState generated by 100 objects, 1.000 objects and 10.000 objects.

In our first test we measure the ViewState sizes sent to a client for each of the three approaches and for three mentioned sizes.First approach is using default settings, which is arguably what many developers do. Second approach is using compression by GZip. It is a custom made approach ([9]) since ASP.NET doesn't support compression out-of-the-box. Third approach uses in-memory Sessionobject to keep ViewState on the server. It uses ASP.NET built-inclass System.Web.UI.SessionPageStatePersister [4]. The size of a ViewState that the client receives is measured simply by taking the whole received HTML and removing all but the ViewState string. The reason for this test is to determine which approach yields the smallest ViewState on client and thus minimizes t_{req} and t_{resp}.

In the second test we take a look at the values for $t_{load}$and t_{save}. The rationale behind this test is to see how those times behave if we minimize t_{req} and t_{resp}. If they remain almost unchanged, we can come to the conclusion which approach gives the best performance. $t_{load}$and t_{save} were measured by using built-in ASP.NET tracing mechanism [4].

The third test was conducted to see how CPU usage behaves in all three approaches and to see if there's expected correlation between $t_{load}$and t_{save}. We used the Performance Monitor in order to see a CPU usage for our web server process. The Web server process was dedicated exclusively to running our tests.

4. RESULTS

Fig. 1. through Fig. 3. show the results of the first test; ViewState sizes that were sent to a client.

We can see that compression significantly reduces the sizes sent to client. For example, if we have 1.000 objects in ViewState, default ViewState size is 675 kb, but if we compress is, the size shrinks to 197 kb, which is almost 3,5 times smaller. As expected, when we use server in-memory Session storage, the ViewState size sent to client is 0 bytes.

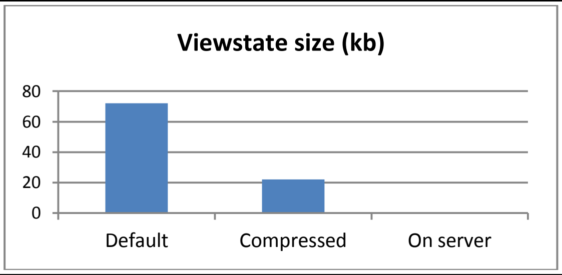

Fig.1. ViewState sizes for 100 objects

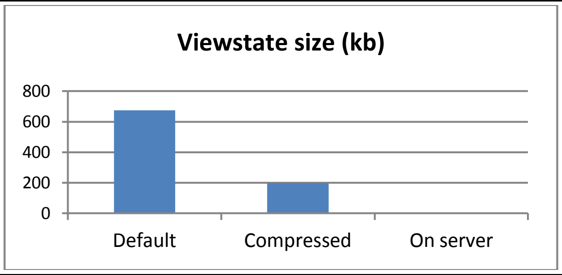

Fig. 2. ViewState sizes for 1.000 objects

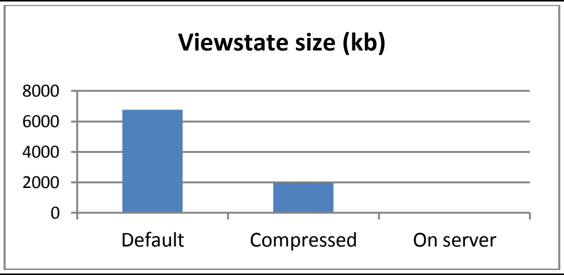

Fig. 3. ViewState sizes for 10.000 objects

Fig.4. through Fig. 6. show the results of the second test; duration of $t_{load}$and t_{save}.

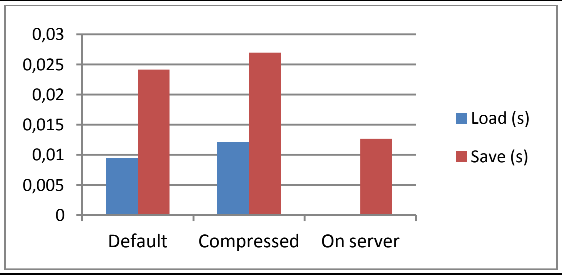

Fig. 4. Duration of duration of t_{load} and t_{save} for 100 objects

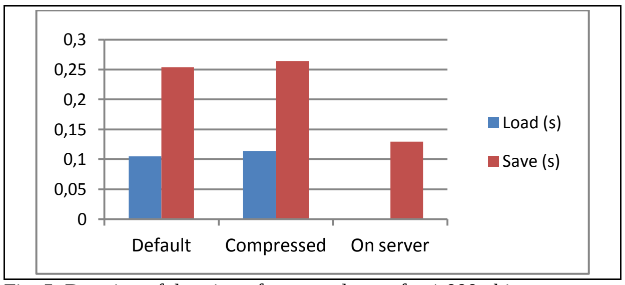

Fig. 5. Duration of duration of t_{load} and t_{save} for 1.000 objects

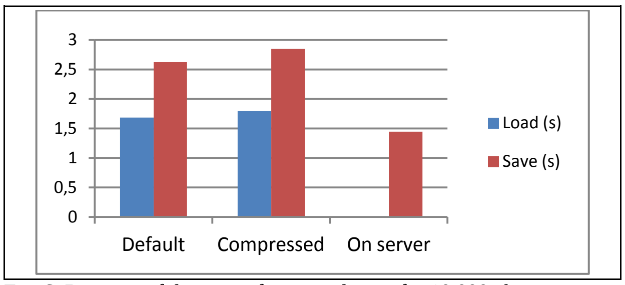

Fig. 6. Duration of duration of t_{load} and t_{save} for 10.000 objects

From last three figures we can see two important things. First, both load and save times rise as we turn on compression, but less that 10%. For example, for 10.000 objects, load time goes up from 1,69 s to 1,79 s (6%). At the same time, save time goes from 2,62 s to 2,84 s (8,4%). This shows that compression requires extra work from server, which is expected.

Second this to note is that load and save times for in-memory Session-held ViewStateare significantly lower. Save times are almost half the sizes of both default and compressed approach and load state is very low (for example, load time for 100 objects is 53 µs). Reasons for that should be sought in the fact that in-memory Session ViewState is not actually being serialized into string.

Fig.7. through Fig. 9. show results of the third test.

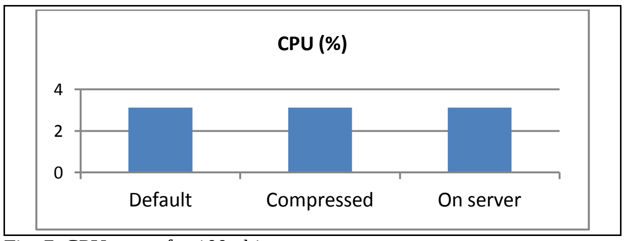

Fig. 7. CPU usage for 100 objects

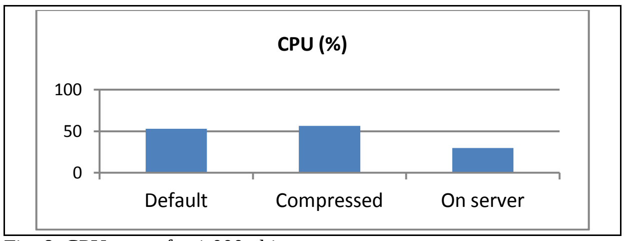

Fig. 8. CPU usage for 1.000 objects

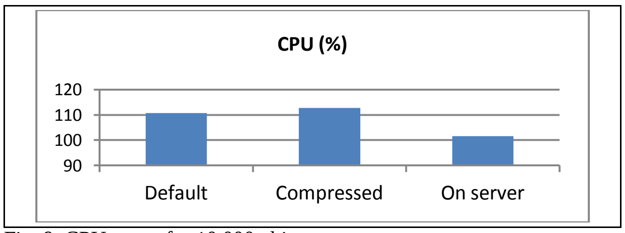

Fig. 9. CPU usage for 10.000 objects

From those figures we can see that the CPU usage goes up a little when compression is used. That is in agreement with the previous figure when load and save times also go up a bit. That means that compression affects CPU usage and delays response generation, but not for much. Expectedly, when leaving the ViewState on the server, the CPU usage is considerably smaller as no serialization takes place.

5. CONCLUSION

We show that using in-memory Session object approach gives the best performance at the price of increased memory usage for Session. But, that approach will not be available if certain conditions are met [10]. Also, if memory usage for Session on the server must be kept on a minimum (in highly concurrent applications) or Session is being held out-of-process or in database, this approach might not be recommended.

The approach that uses compression shows good performance at the price of slightly increased CPU usage on the server. If that is not an issue, this approach is recommended. One weak point of it might be non-existent built-in support for it. Also, if the entire Web form uses compression at a HTTP level, this approach is redundant and should not be used.

Finally, we show that the default approach suffers from potentially very large ViewState transferred to and from the client. This can severely affect the performance and user experience, so this approach should be used as the last resort.

6. REFERENCES

[1] http://w3techs.com/technologies/overview/programming_language/all,Usage of server-side programming languages for websites, *Accessed on: 2012-08-15*

[2] http://www.alexa.com/topsites, Alexa Top Global Sites, *Accessed on: 2012-08-15*

[3] http://www.asp.net/web-forms, Web Forms : Official Microsoft Site, *Accessed on: 2012-08-12*

[4] http://msdn.microsoft.com/en-us/magazine/dd942833.aspx, Comparing Web Forms And ASP.NET MVC, *Accessed on: 2012-08-13*

[5] http://codebetter.com/karlseguin/2010/03/11/webforms-vs-mvc-again/, WebForms vs MVC, *Accessed on: 2012-08-14*

[6] http://w3techs.com/technologies/details/pl-aspnet/all/all, Usage statistics and market share of ASP.NET for websites, *Accessed on: 2012-08-14*

[7] Stojanovski, T.; Vuckovic, M.; Velinov, I. (2012). Empirical study of performance of data binding in ASP.NET web applications, *Available from:*http://arxiv.org,*Accessed:* 2012-08-16

[8] Stojanovski, T.; Vuckovic, M.; Velinov, I. (2012). Scalability of Data Binding in ASP.NET Web Applications, *Available from:*http://arxiv.org,*Accessed:*2012-08-16

[9] http://www.bloggingdeveloper.com/post/How-To-Compress-ViewState-in-ASPNET-20-ViewState-Compression-with-SystemIOCompression.aspx, Compress, *Accessed on: 2012-09-07*

[10] http://davidovitz.blogspot.com/2006/04/things-that-you-should-watch-out-when.html, Things that you should watch out when using SessionPageStatePersister, *Accessed on: 2012-09-08*