A Model-View-DynamicViewModel and its Performance in a Web-based Component Architecture

Graeme Baillie, Brian Armour, Dave Allan, Robert Milne
CC Technology Ltd
Glasgow, Scotland
{graeme.baillie, brian.armour, dave.allan, robert.milne}@cctechnology.com

Thomas M Connolly, Richard Beeby
School of Computing
University of the West of Scotland
Paisley, Scotland
{thomas.connolly, richard.beeby}@uws.ac.uk

ABSTRACT

The implementation of an appropriate software architecture is crucial in achieving the optimum performance from a system. Web-based applications are becoming increasingly popular in replacing previous Windows-based applications. This has lead to a growth in new technologies and architectures to cope with the new workload and performance demands that web-based applications can require.

This paper will look at a current software company that provides process-heavy web-based application systems providing grant management services to medical and academic research. The purpose of this paper is to analyze the performance of their existing Web Forms based management system against newly developed dynamic component-based architecture using a variant of the Model View Controller (MVC) and Model-View-ViewModel (MVVM) patterns, which we have termed MVDVM (Model-View-DynamicViewModel). Accurate and meaningful results will demonstrate how applications can now benefit from this approach.

Keywords – Component based architecture, Dynamic forms, Interoperability, Model View Controller.

I. INTRODUCTION

Web-based software applications are becoming more popular and crucial to the running of businesses in their day-to-day processes. Due to the growth of the world wide web and of Internet services capabilities, software companies are in a much better situation to offer services that previously would have only been possible through an offline platform.

Improvements in Internet connectivity and hardware performance within the last ten years has meant that software developers no longer have to worry as much about bandwidth, processor or memory requirements when implementing process-heavy applications. Although this has been beneficial to the software industry it has lead to many systems being implemented in ways that do not utilize best performance techniques available. This can deliver sub-optimal user experience and utilizes greater server capacity than is necessary at a time when datacenter costs are rising.

This paper will look at a software company that implements process-heavy web applications. The company’s current working practice and techniques being used will be analyzed. The techniques will then be compared against a new dynamic component-based solution to identify the change in performance achieved for one of its key software products.

Statistics such as page load time, page size, number of database queries will be measured under varying loads and analysed to assess how these will benefit by applying latest thinking and best practice when developing high-performance software systems.

In the next section, a brief overview of the product is provided followed by a discussion of related research. The subsequent sections will discuss the technology platform, the test platform and then present the performance data of the old product and the re-architected product.

II. CASE STUDY OVERVIEW

This paper analyzes a component-based dynamic solution used to manage application forms within CC Grant Tracker, a Grants Management software product. The solution builds and publishes forms, which are then used for the completion, submission and ongoing management of applications. CC Grant Tracker is a solution from CC Technology, a leading global supplier of advanced grant management software solutions, based in Glasgow, Scotland. CC Grant Tracker supports organizations such as charities, academic institutions, public funding bodies and corporate foundations who often employ a regular cycle of grant applications, where professional reviewers decide which applications are worthy of funding. For a large organization, there may be hundreds or even thousands of applications, as well as dozens of reviewers, not to mention stringent auditing requirements. In such cases, simply administering the application process is a major undertaking. CC Grant Tracker manages the full life cycle of grant administration, from initial application through evaluation, approval and ongoing management.
When CC Technology originally created the software product the emphasis of the initial analysis and design phase was based on functionality rather than the performance of the system. The application is heavily web forms-based and the initial version was delivered using the standard web development tools from Microsoft – ASP.NET Web Forms. This delivered a tightly-coupled solution with associated maintenance issues. As the number of clients and the breadth of their requirements have increased the requirement to develop the flexibility and performance of the application forms has become a priority.

CC Technology have been using a re-architected solution with a number of new large clients and the new system has been extremely well received.

III. RELATED RESEARCH

A. Web Forms vs MVC

Microsoft currently endorses two web architectures, Web Forms, which was released in 2002, and the Model-View-Controller (MVC), which was released in 2009. MVC is a standard design pattern that Microsoft has adopted. The MVC pattern helps deliver solutions that separate the different aspects of the application (input logic, business logic and UI logic), while providing a loose-coupling between these elements [1]. Both architectures provide benefits to the developer and end-user depending on the circumstances. For this study we examined both approaches to obtain a better understanding as to which provides the better performance benefits.

1) Web Forms – The major advantage of a Web Forms implementation is that it allows an application to be developed in a relatively short period of time as there is lower initial design requirements for the solution architecture. However, research and experience shows that there are a number of performance disadvantages inherent to this approach.

Web Forms allow only one form tag to be added per page. This means that in order for the page to get information to or from the server, the entire page must be posted back. A common approach to address this uses Microsoft’s Ajax Control Toolkit and update panels. This reduces performance problems related to sending unnecessary data back to the server however the Ajax update methods can be just as performance heavy in bandwidth and processing on the server due to the View State that is required on all updates from a Web Form regardless of a full postback or asynchronously.

The View State holds session information for each page [2]. This adds considerably to the data size of the page and as this is exchanged in all updates this impacts the bandwidth for the solution. Many of the built-in ASP.NET components make heavy use of View State, so it is not unusual for a page to have 10’s of Kbytes of View State [3]. This is measured in terms of performance for end-users and bandwidth costs at the data centre. There is therefore no performance gain in using Ajax update panels throughout the application as the net effect can be the same or greater than doing a full postback.

A further serious issue for modern web pages is the abstraction from HTML, as this hinders accessibility, browser compatibility, and integration with JavaScript frameworks like JQuery and PrototypeJS [4]. Of lesser importance to this case study, the postback model makes it harder for search engines to rank ASP.NET pages high.

2) MVC – The Model-View-Controller (MVC) architectural pattern was first proposed by Trygve Reenskaug in the late 1970s (the earliest source available is [5] but MVC was not publicly documented until 1988 [6]). It was first used in Smalltalk [7] and is an effective approach for supporting multiple presentations of data. Users can interact with each presentation in a style that is suitable to the presentation. The data to be displayed is encapsulated in a model object and each model object may have a number of separate view objects associated with it, where each view is a different display representation of the model. Each view has an associated controller object that handles user input and device interaction. MVC was devised to target desktop applications, but because of its relatively loose formulation it was easily adapted for the Web. MVC (or Model 2 using the Sun Microsystems terminology [8]) is now used widely in web frameworks (e.g., ASP.NET, Django, Ruby on Rails, Code Igniter, Apache Struts, JavaServer Faces). In Web MVC (which differs somewhat from traditional MVC), a model encapsulates application data in a way that is independent of how that data is rendered by the application. The view accepts some number of models as input and transforms them into appropriate output that will be sent to the browser. The controller connects the models and views, typically by gathering model data and sending it to the view for rendering. The view accepts data as input and produces a string as output, which may include information about its type (e.g., HTML, XML or JSON). After being manipulated by some post-processing filters, the string is sent directly to the browser. Because the view’s output is treated as an opaque string, it is difficult for the framework to reason about the structure of the content. These views are complicated by the need to provide both Ajax and non-Ajax versions of a site [9]. In ASP.NET, MVC is implemented as shown in Figure 1(a).
In addition to managing complexity, the MVC pattern makes it easier to test applications than it is to test Web Forms. For example, with Web Forms a single class is used both to display output and to respond to user input, which when testing has to be instantiated along with all its child controls and additional dependent classes. Moreover, tests in Web Forms require a Web server. In contrast, the MVC framework decouples the components and makes heavy use of interfaces, which makes it possible to test individual components in isolation from the rest of the framework. The loose coupling between the three main components of an MVC application also promotes parallel development. For instance, one developer can work on the view, a second developer can work on the controller logic, and a third developer can focus on the business logic in the model.

A variant of MVC is the Model-View-Presenter (MVP) pattern [10], as shown in Figure 1(b). In MVP, the Presenter has the same responsibilities as MVC’s Controller, acting as a mediator between the view and the model. It receives user input requests from the view and evokes changes on the model in response. The Presenter is able to query the view for data but to enforce separation of concern (SoC), the presenter is decoupled from the view by holding a reference to the view’s interface rather than its implementation [11] [12]. Fowler describes a different approach for achieving SoC called the Presentation Model [13]. This pattern is similar to MVP in that it separates a view from its behavior and state and introduces a Presentation Model that is an abstraction of a view.

A further variant of MVC is the Model-View-ViewModel (MVVM) design pattern for Microsoft’s Windows Presentation Format (WPF) [14], as shown in Figure 1(c). The MVVM pattern is a more specific version of the Presentation Model pattern [21]. MVVM also separates the view from the logic. However, unlike Controllers in MVC (Figure 1a) and Presenters in MVP (Figure 1b), an MVVM ViewModel has no awareness that a view even exists [15]. The ViewModel is an abstraction of the view but does not need a reference to the view like MVP does. The view uses the ViewModel as a data context and binds properties to fields in the ViewModel, providing a very loosely coupled design. This separation allows a graphics design team to focus on the view while a software development team can focus on implementing a stable and good ViewModel. As the ViewModel does not have to reference the view, the logic can be tested without the view and it is also easy to make different views for GUI evaluations and compare them without changing the ViewModel. Note, Esposito and Saltarello [16] regard MVVM as the same pattern as Fowler’s Presentation Model.

Zeller and Felton [9] introduce a stateless, framework-agnostic web application development style, which they call SVC (Selector-based View Composition) (SVC). With SVC, a developer defines a web page as a series of transformations on an initial state where each transformation consists of a selector (used to select parts of the page) and an action (used to modify content matched by the selector). SVC applies these transformations on either the client or the server to generate the complete web page. The authors contend this approach has two advantages: (i) SVC can automatically add Ajax support to sites, allowing developers to write interactive web applications without writing any JavaScript; (ii) developers can reason about the structure of the page and produce code to exploit that structure, increasing the power and reducing the complexity of code that manipulates the page’s content.

There are many other approaches to SoC. Presentation-Abstraction-Control (PAC) [17] in which the UI is formed by a tree of agents for each of the three components (Presentation, Abstraction and Control). In Naked Objects [18], a UI is automatically generated from the model by analyzing the model interface using Java’s reflection features. As a result, the model must contain all of the application domain logic. On the other hand, in Visual Proxy [19] the model/view separation is abandoned. While the model and view objects are still separate objects, they are implemented within the scope of a single class. The model and view layers are tightly coupled, rather than decoupled. Holub argues that the purpose of a model is to provide services to the view and if they are separated, the model cannot provide all the services for the view.

IV. TECHNOLOGY PLATFORM

CC Technology has a proven history in the use of Microsoft development tools and therefore it was logical to deliver the original system using ASP.NET Web Forms which was the prevalent Microsoft technology in 2004 when the solution was implemented. The Microsoft Ajax Control Toolkit was implemented to limit the amount of postbacks needed to the server. For database connections a custom ADO.NET entity generator connects to a SQL Server database. This platform has worked well, the technologies interact successfully and developers can develop and maintain a web-based solution without having detailed low-level web knowledge.

The new component-based approach uses what we have named an MVDVM (Model-View-DynamicViewModel) architecture. The MVDVM architecture is an MVC-based architecture that is heavily influenced by the MVVM (Model View View-
Model) architecture, as Figure 2 shows. It contains a set of view-models and handles each one in a similar way. The major difference between the MVVM and MVDVM architectures is that the latter model dynamically builds and persists the view-models at runtime rather than having statically designed view-models, as in MVVM. The architecture is extremely powerful because of the ability to have encapsulated components (view-models) that persist themselves, thereby requiring no code to save and load components, producing an architecture that is extremely extensible. Each component that is added has no dependency or relationship to other components, allowing a separation of concerns and removing the risk of breaking other components when introducing new ones. By having this ability the only development that has to occur is writing new view-models. The view-models’ logic and behaviour are captured in the view, which means that they can be updated or created without the need to compile. This in turn allows role-based logic to be performed on smaller subcomponents than would usually be possible. The view-models can be any user control such as text boxes, drop down lists or more complex composite user controls with contained business logic. A component can also seamlessly interact with other non-MVDVM systems and other databases meaning that saving and loading is not limited to a component’s own data. The architecture allows the user to specify at runtime HTML to render each individual component to multiple document types to allow hard copies of forms. The difference between what the MVDVM architecture offers from other form designer tools and architectures is that it has no bounds or limitations as to what the components can do, other than what the .NET framework offers. The entire framework is self contained, which allows seamless integration with other architectures and systems. The scalability of controls and ease of extensibility make this architecture a good alternative to the web forms implementation.

The architecture is also advanced through use of client-side logic in JQuery components. As JQuery is now part of Microsoft’s release with MVC, the integration in the development environment is seamless and therefore easy to use. Having JQuery allows quicker and more efficient development environment is seamless and therefore easy to use. The major change to the technologies used was the move away from the company’s own entity generation tool for Database access. A repository pattern was implemented into the software so that at any time we could change the database connection technique without consequence. Initially Microsoft’s Entity framework was evaluated. However, due to the early version of the software it was decided that the more established and Nhibernate model should be used.

V. TEST PLATFORM

To ensure accurate measurement of the performance of the original Web Forms and the new MVDVM architecture a common set of tests were developed. The tests were performed under the same situations and with the same metrics. A number of standard web pages in each system were tested with single users and also with increasing number of users (load testing). Both tests resulted in page load speed, number of pages per second and database information being obtained.

An accurate and consistent set of results was achieved by running a dedicated environment comprising a LAN, a test client containing load testing software, a web server running an instance of the forms software and a database server. To ensure that hardware performance was not compromising the results, the memory and CPU usage were monitored on each computer to ensure they stayed within 60% utilization. MS Visual Studio 2010 Ultimate Edition Load testing suite was used to generate relevant graphs and performance data.

VI. SERVER PERFORMANCE

Individual tests were carried out on both architectures. The test cases used pages on each system with the same function and content, albeit with different implementations.

<table>
<thead>
<tr>
<th>Area</th>
<th>Page Load Time (Average(seconds))</th>
<th>Time to receive first byte (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opening a form</td>
<td>0.413</td>
<td>0.145</td>
</tr>
<tr>
<td>Financial page</td>
<td>1.361</td>
<td>1.113</td>
</tr>
<tr>
<td>Standard page</td>
<td>1.744</td>
<td>0.668</td>
</tr>
<tr>
<td>Saving form (including redirect)</td>
<td>0.635</td>
<td>0.588</td>
</tr>
</tbody>
</table>

Figure 3: MVDVM Page Times (in seconds)

<table>
<thead>
<tr>
<th>Area</th>
<th>Page Load Time (Average)</th>
<th>Time to receive first byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>Opening a form</td>
<td>1.526</td>
<td>0.940</td>
</tr>
<tr>
<td>Financial page</td>
<td>1.556</td>
<td>0.599</td>
</tr>
<tr>
<td>Standard page</td>
<td>1.744</td>
<td>0.668</td>
</tr>
<tr>
<td>Saving form (including redirect)</td>
<td>0.656</td>
<td>0.114</td>
</tr>
</tbody>
</table>

Figure 4: Web Forms Page Times (in seconds)

The performance results in Figures 3 and 4 clearly show that the MVDVM architecture is faster on an individual load. It must be noted that there is a change in logic between the two solutions. The MVDVM application forms bind the models dynamically on every page load. Thus for every load and page exit data is read and written to the database. The Web Forms...
solution does not write to the database until a user hits submit, which may be after n*page visits. This is reflected in the load times for the financial page. This is a complex page where hundreds of requests are being sent to the database. Although the MVDVM is still faster for this page the number of connections is the reason the difference is not as substantial.

The single tests are a good indication of the efficiency of each architecture. However, live systems are characterized by uneven and high loads. For a production system the number of concurrent users could be in the hundreds. Load testing was carried out on the same pages as noted in Figures 3 and 4. The resulting graphs are shown in Figures 5 and 6 (Vertical axis represents number of users, number of pages per second and average page load per second and all using corresponding ranges outlined in figures 7 and 8). The graphs show that the MVDVM out performs the Web Forms by a large margin. Each test began with one simulated user running a web test. This proceeded through each page in a defined order. Every ten seconds five more users were added to the current list of users. The results are shown in Figures 7 and 8 for each run of the test.

<table>
<thead>
<tr>
<th>Range on graph</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>User load (num users)</td>
<td>1000</td>
<td>1</td>
<td>176</td>
</tr>
<tr>
<td>Pages/sec (seconds)</td>
<td>10</td>
<td>0</td>
<td>7.8</td>
</tr>
<tr>
<td>Ave page load (seconds)</td>
<td>100</td>
<td>0.25</td>
<td>39.7</td>
</tr>
</tbody>
</table>

**Figure 7: Web Form Graph Averages**

<table>
<thead>
<tr>
<th>Range on graph</th>
<th>Min</th>
<th>Max</th>
<th>Avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>User load (num users)</td>
<td>1000</td>
<td>1</td>
<td>176</td>
</tr>
<tr>
<td>Pages/sec (seconds)</td>
<td>10</td>
<td>1</td>
<td>39</td>
</tr>
<tr>
<td>Ave page load (seconds)</td>
<td>100</td>
<td>0.35</td>
<td>13.9</td>
</tr>
</tbody>
</table>

**Figure 8: MVDVM Graph Averages**

Figure 5 shows that the average page load speed increases in relation to the number of users. As the number of users increases the number of pages loaded per second severely decreases.

The MVDVM results are quite different. The pages per second start very high and remain so throughout the test. At the points of lowest performance where the graph dips the response times are still acceptable for an end-user. These dips occur where a large number of users (more than 100 users) tried to call an action concurrently. This is a quirk of using a test suite and the phenomena are unlikely to occur in production at this usage level. Usually in this case it would be the hardware performance that would fail or cause performance issues. The page load speed also remains very constant and, apart from the dips, remains lower than one second for each page throughout the test.

**VII. CLIENT PERFORMANCE**

Client performance can be characterized by the download time for the pages and of the rendering time.

Figure 9 shows the details for an average page in the performance tests. The details emphasize the earlier comments about how large or inefficient the HTML can become due to the View State and the ASP.NET automatically generating HTML. The Javascript size is also very high as the previous forms were rendering pure Javascript. The Javascript was not compressed, which is also inefficient, causing larger page sizes.

**Figure 9: Web Forms Average Page Detail**

**Figure 10: MVDVM Average Page Detail**

The detail of an average MVDVM page in Figure 10 shows that the page size is smaller. The ability to control the HTML and use JQuery to write minimal amounts of code has resulted in a reduction of the page size from 471k to 176k.

The HTML code in the Web Forms output is controlled by ASP.NET and is not standards-compliant, whereas the MVDVM output is controlled by the developer. This control allows the developer to address client performance issues, which Google describe as 5 performance factors [20]: (i) optimizing caching, (ii) minimizing round-trip times, (iii) minimizing request overhead, (iv) minimizing payload size, (v) optimizing browser rendering. The MVDVM solution improves on each of these factors through the communications and page content.

**VIII. DATABASE PERFORMANCE**

A statistical analysis of the database transaction times compares the performance between the two architectures. The results are split into two parts: using all the data and using only the data that had duration times greater than zero. The results show every query that happened when the tests were run. Because of this there are many light weight queries that had durations of less than 1 ms, these show as 0ms durations.

**A. Analysis using all of the data**

Using the Web Forms architecture, reads were performed an average of 48.60 times (SD = 1061.05) with a range of 0 to 32,350 whereas using the MVDVM architecture reads were...
performed an average of 24.54 times (SD = 283.12) with a range of 0 to 6,440.

Using the Web Forms architecture writes were performed an average of 0.0145 times (SD = 0.30) with a range of 0 to 17 whereas using the MVDVM architecture writes were performed an average of 0.0093 times (SD = 0.20) with a range of 0 to 22.

Transactions using the Web Forms architecture had an average duration of 4.729 ms (SD = 20.55) with a range of 0 to 93 ms.

Paired samples t-tests indicated that the MVDVM architecture performed significantly less read commands (t(65,527) = 5.607, p < 0.000), significantly less write commands (t(65,527) = 3.733, p < 0.000) and took significantly less time to perform read and write commands (t(65,527) = 4.729, p < 0.000) than the Web Forms architecture.

B. Analysis using only the data that had duration times greater than zero

Using the Web Forms architecture reads were performed an average of 2,047.19 times (SD = 6,955.46) with a range of 0 to 32,350 whereas using the MVDVM architecture reads were performed an average of 990.40 times (SD = 1,664.82) with a range of 0 to 6,440. Using the Web Forms architecture writes were performed an average of 0.1095 times (SD = 0.20) with a range of 0 to 17 whereas using the MVDVM architecture writes were performed an average of 0.0093 times (SD = 0.20) with a range of 0 to 22.

Transactions using the Web Forms architecture had an average duration of 0.1223 ms (SD = 2.04) with a range of 0 to 22.

Paired samples t-tests indicated that the MVDVM architecture performed significantly less read commands (t(1,406) = 5.339, p < 0.000), significantly less write commands (t(1,406) = 5.996, p < 0.000) and took significantly less time to perform read and write commands (t(1,406) = 4.764, p < 0.000) than the Web Forms architecture.

IX. Conclusion

This paper has presented a comparison of two web architectures used in process-heavy web applications. The first, Web Forms, is an established standard Microsoft approach and the second, MVC, is a relatively recent approach gaining some acceptance in the development community. This paper has proposed a variant of MVC, called MVDVM and has demonstrated that this architecture delivers performance benefits over the traditional Web Forms based architecture. The performance has been shown to be improved at the database, web server and client. With each of the tests it has been shown to be faster or more efficient.

The results of the case study have proven that under these circumstances the performance benefits are large and the move to an MVC architecture are justified.

X. Acknowledgements

This project received financial support from the Knowledge Transfer Partnerships programme (KTP). KTP is funded by the Technology Strategy Board.

XI. References