

Risetime

ECM • BPM • Web Solutions



The Challenges of Integrating Silverlight 2.0 into existing ASP.NET Applications

A White Paper for Technical Users

Contributing Authors:

Dan Sinclair
Mike Federico
John Hauppa

547 West Jackson Blvd. 8th Floor
Chicago, IL 60661
P 312.362.9930
F 312. 362.9925

info@risetime.com



www.risetime.com

Table of Contents

- Introduction 1
- An Iterative Approach 1-2
- Web Services 3-4
- Error Handling 4
- ViewState, Session, and Configuration 5
- Serialization 5-6
- Integrating Validation 6
- Globalization/Localization 7
- Handling Post Backs 7-10
- Conclusion 10

Introduction

The introduction of Microsoft Silverlight 2 has created a new paradigm of business application development, merging the economy and commodity of server based web applications with the rich user experience and ease of development of client-side applications. This new model allows developers and designers to work together to rapidly prototype designs and layout for customer approval, separating user interface concerns from implementation making collaboration easier than ever before. Silverlight incorporates rich media elements, full customization of UI components, and cross-browser compatibility unequaled in any other Microsoft web solution. Silverlight is positioned in the Microsoft lineup as becoming the future of web development.

Many millions of lines of code currently exist in production business applications written in ASP.NET. Microsoft based web applications have become a *de facto* standard for every facet of e-business, from simple shopping and order processing sites, to line-of-business software-as-a-service applications that run large corporations. ASP.NET has become so prolific because it works. It's large suite of built-in components and integrations with other Microsoft and industry standard products as well as integrations supported by the thriving ASP.NET developer community give it a marked advantage over other products in the same space. The .NET base class libraries which comprise the foundation of ASP.NET allow rapid access to technologies such as Active Directory, SQL Server, and Windows Management Instrumentation (WMI) while providing security, internationalization support, and network interconnectivity effectively transparent to the development team.

Microsoft's new Silverlight framework is an excellent candidate for re-architecting existing ASP.NET solutions during maintenance cycles, providing the ability to iteratively include Silverlight components which interact seamlessly with existing ASP.NET code. These components make available the rich user interface of Silverlight and allow existing ASP.NET applications be re-engineered for a better design. Implicitly, Silverlight's architecture promotes a tiered, service oriented architecture, utilizing web services to talk to server-side business logic. This means that active, production projects can be converted over to using Silverlight as time and budget allows, slowly incorporating richer application features without drastically increasing project scope. This article covers some of the frequent pain points and obstacles to incorporating the new Silverlight technology into existing ASP.NET applications.

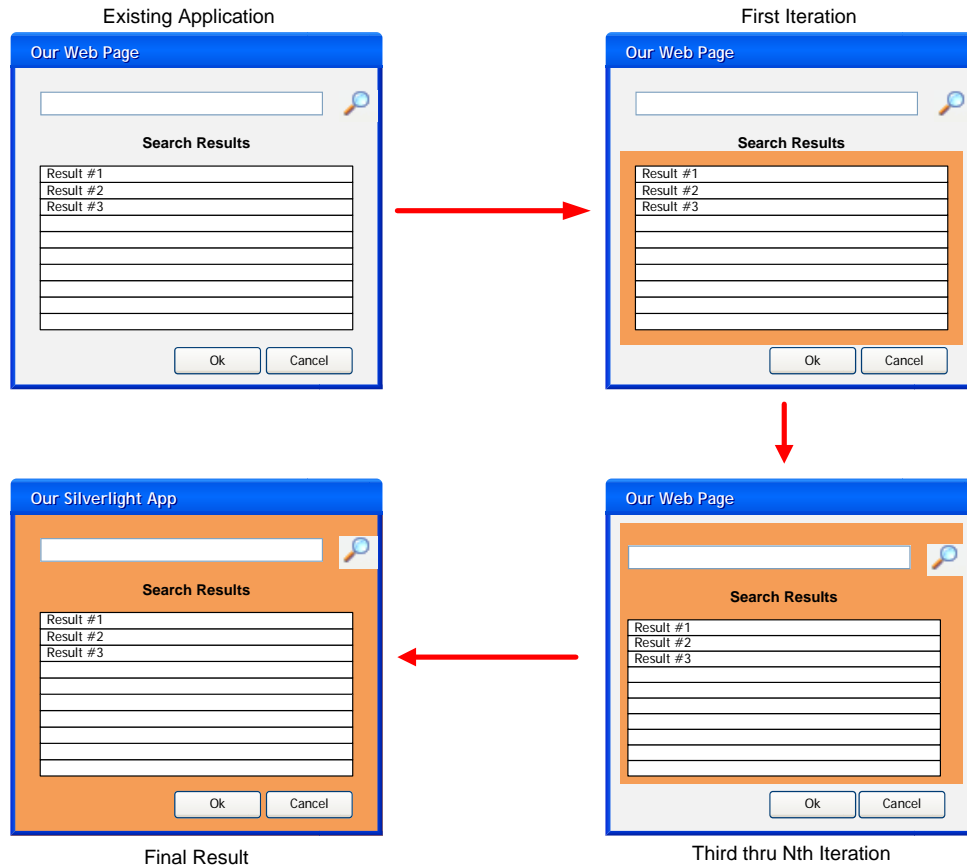
An Iterative Approach

Developing a project roadmap for integrating Silverlight into an existing application can be a daunting task. It is important to balance risk with ensuring that the project is adding business value. The best method for most types of applications is to follow an iterative approach: first designing, developing, testing, and delivering small portions of the application based on minor enhancements, followed by larger and larger sections. The first iteration can be used to generate excitement and buy-in from business users by re-architecting some of the user interface controls and page sections which cause users the most pain. This pain can be caused by long post-backs forcing users to wait for the page

to reload, limited searching and sorting functionality, limitations working with large datasets, or any number of user interface quirks inherent to basic ASP.NET applications.

As these sections are developed and integrated with the application, the individual controls can be grouped into larger and larger sections as more of the page becomes Silverlight controls. As the Silverlight controls grow, so does the separation of concerns between the user interface which comprises the Silverlight component, and the business logic and data access code which reside on the server. This forced tiered design helps older applications slowly transition from a page based procedural design to a service oriented architecture. For example, users report pain interacting with a search results grid in an ASP.NET application when working with large data sets. This component can be redesigned in Silverlight, added to the existing application, and populated from a web service based on the original code used to populate the ASP.NET search grid. This fix provides a solution to an immediate user pain, and could be followed by redesigning the search control, and finally, in a third iteration, tying together the two controls into one Silverlight search component. This single component could be further enhanced by adding some of Silverlight's new features to improve the user experience, such as customizable search preferences in the search component which could remember the user's previous searches and settings and store it in Isolated Storage on their local machine.

Typically, a fraction of the development cycle will be wasted in designing code that helps to allow the Silverlight controls to interact with the ASP.NET application. Time spent building shim code and other integration workarounds to leverage the Silverlight components in the existing application during these iterations would be offset by the amount of testing these components receive as part of the development lifecycle. This reduces the delivery time for the completed product, easing concerns that a monolithic deployment of code would introduce or reintroduce bugs into the application. Eventually after many development cycles, every individual page in the application will be architected in Silverlight, and these pages can be integrated into a single logical Silverlight application which exists on a single ASP.NET page. This would allow the entire application to be streamed down to the user once when they first accessed it, and the user interface could stay consistent and responsive even during context changes, greatly improving user experience. Because the Silverlight application is run client side, using web services to communicate with server resources, the application's implementation develops a tiered structure.



Iterative Approach to Include Silverlight

Web Services

Web services are Silverlight's connection to the outside world. They allow client-side Silverlight applications to interact with code and resources which exist on the server from which Silverlight is hosted, as well as environments which have been properly designated to be accessed from Silverlight. There are three basic flavors of web service access that are available in Silverlight. First and most desirable, are Windows Communication Foundation (WCF) web services. WCF is Microsoft's latest design for a unified framework of application communication based on a service oriented architecture (SOA) design. Silverlight natively consumes WCF web services, though it does not currently implement all of the functionality of WCF. Silverlight lacks support for FaultContract, a major component of WCF, which leaves implementation of fault handling up to the development team. Silverlight also can only consume the most basic mode of WCF web services, the basic HTTP bindings in standard ASP.NET compatibility mode. This severely limits the power of WCF in interacting with a Silverlight application, but could prove useful in the context of upgrading existing applications to Silverlight. The ASP.NET compatibility mode allows the WCF web services to access ASP.NET session, request, and response information that would be unavailable to higher order WCF services, but could be useful in re-architecting applications that heavily rely on session access. This is not to say that future releases of

Silverlight will not improve the support for WCF, and upon upgrading to proper use of WCF and re-architecting the entire application in Silverlight on top of web services, the offending code which accesses the session information could be factored out.

The second method of communication for Silverlight is the ASMX SOAP web services which were the standard for ASP.NET until the introduction of WCF in 2006 and the only web services available for ASP.NET 1.1 applications. ASMX web services are much simpler in design to WCF, and are generally well supported by Silverlight due to its basic SOAP format. Older applications which support a web service based API will generally have services of this type. These services execute in context of the ASP.NET application, and therefore code which calls this service has access to the ASP.NET session, as well as any other application services that are available to the current ASP.NET based page code.

Another approach to communication for Silverlight would be to utilize REST services. REST services are different than WCF and ASMX services in that they generally involve a loose communication structure, relying on URL structure to determine the relationship between data and operation. While WCF and ASMX web services are interface-based, generating client-side code which can be auto-generated based on the service implementation, REST services are generally implemented using the WebClient and XmlSerializer classes to make web requests and read the responses back from the server using the HTTP verbs to control actions which occur upon objects. Since there is no strict object typing in REST web services, these services are easily consumed by Silverlight, though this also means all of the implementation is left up to the developers, from error handling to ensuring the correct format of the result data.

By far the best approach if available (2.0 or later applications), would be to use the WCF web services, because even in their basic mode, they provide more functionality and extensibility than the other two types of services. WCF services can provide the binding functionality compatible with Silverlight, and also expose more complex bindings which could interact with other consumers, or with later versions of Silverlight.

Error Handling

Handling web service errors and returning client side error reports to the server are integral parts of a well-designed client side application. As part of the WCF service stack, the FaultContract is used to generate a contract of possible errors that a service can expect in normal operations, allowing the service to marshal errors to the service consumer which it can understand. Silverlight, unfortunately, does not support FaultContract (in fact, Silverlight currently has problems consuming services that include FaultContract). This means that when marshalling errors between the server and client, another method must be used.

A flexible approach is to include an “out” parameter as part of all web service calls which contains the error information. It contains the error message and exception information as part of the resulting out value. This should not include stack trace information from the server, as this could create a security concern exploitable by nefarious users. This out parameter could also be used to store other

operational information regarding the application similar to the IExtensibleDataObject interface which Silverlight also does not support.

ViewState, Session, and Configuration

Access to server-side and client-side session and configuration information is integral to almost every ASP.NET application. The client-side session information stored in the ViewState is potentially readable by Silverlight, although no built in support is included in the Silverlight framework. Since the ViewState store generates a tamper-proof hash based on the machine key of the server, and can be potentially encrypted, writing to the ViewState from Silverlight is not possible unless security is lowered. Removing the tamper-proof hash from the ViewState can be accomplished by setting the value of enableViewStateMac to false on the Page object in ASP.NET. Accessing and/or modifying information about a page ViewState from Silverlight is technically possible given these constraints; it should not be used as a general practice. There are a few circumstances where examining the ViewState from Silverlight might be useful, such as handing state changes from a post back when the Silverlight stays resident on the page.

Session and Configuration information stored on the server present a completely different issue. Exposing these settings via a web service or similar mechanism could easily create a security hole. Since Silverlight code is client-side, it is easily decompiled by a nefarious user and any security mechanism in place to prevent setting access could be compromised. Access to these settings is best hidden behind purpose built web services which provide data objects based on the session and configuration.

Serialization

Serialization of objects between Silverlight and ASP.NET is probably the largest discontinuity between the two environments. ASP.NET provides a more robust heavyweight serialization library, offering a multitude of XML serialization formatting options, binary serialization, as well as WCF data contract serializers to facilitate WCF web services. Silverlight, on the other hand, is designed only with lightweight, basic serialization support in order to reduce code overhead and make deployment and cross-browser support more practical. The most glaring problem with Silverlight's serialization library is the complete lack of support for the ISerializable interface, which allows any type to control its own serialization. Silverlight also does not support the Serializable attribute, so this means that many types which are freely XML serializable in the full .NET framework are unavailable for serialization in Silverlight.

Two other drawbacks which hamper Silverlight's ability to integrate with other parts of the .NET and WCF software stacks are the lack of a binary formatting serializer, as well as not implementing the NetDataContractSerializer class. The lack of these two serializers is understandably by design, but many existing web services and pages use these technologies. These limitations prevent the marshalling of data objects between Silverlight and ASP.NET. This is a problem especially if attempting to consume web services which marshal types across web service boundaries or use in a remoting scenario. Because of security limitations in Silverlight's reflection library which prevents access to non-public members to

any non-class callers, creating a binary serializer by hand for Silverlight would be an essentially impossible task without designing types specifically for this formatting. The `NetDataContractSerializer` is similar to the `DataContractSerializer` which is implemented in Silverlight except that it additionally marshals type data. This would mean that it could effectively generate and populate a type which exists on both the client and server. Removal of the functionality of these serializers was to prevent the making of bad design decisions, but their lack can prevent considerable interoperability with existing applications.

Integrating Validation

ASP.NET supplies developers with a rich suite of validation features that make building form data and business logic validation a near-trivial task. As part of the first release of Silverlight 2, Microsoft did not include any validation features beyond the implicit validation used by controls to reduce user errors, such as the calendar control preventing invalid dates. There are still times in which validation is required for both business logic and field validation for specific business needs. Generating validation code for Silverlight controls and alerting users to validation problems and how to eliminate them is a task left up to the development team. Generally speaking, this involves designing a validation framework manually, both building a method by which Silverlight can identify errors as well as returning them to the user in a meaningful way.

Integrating Silverlight into an existing ASP.NET application which uses ASP.NET validation presents a second problem. Making Silverlight validation errors which happen client side, integrate with the ASP.NET validation summary, which is populated server side, is a challenge. The simplest approach would be to add a hidden field to the form which after the Silverlight control executes its validation; it sets the value of the hidden field to blank if there were no errors, and the value of the error summary information if there were errors. On the server, an ASP.NET `CustomValidator` control is created which reads this field and creates a validation error if the field is non-empty. The validation error message component would be the value of the hidden field. The Silverlight component could alert the user to problems with its component, and along with the ASP.NET page prevent the user from continuing until the error has been fixed.

A second concern regarding validation is rewriting validations from existing ASP.NET server controls which are being rewritten in Silverlight. The restructuring can sometimes become confusing since Silverlight code runs client-side, and ASP.NET validation occurs as a loose conglomerate of client and server side validation methods. Often restructuring a specific validation in Silverlight may require adding a web service to provide lookup values from a database or other server based resource. A validation framework for Silverlight which performs validation on the server and allows Silverlight controls to simply submit the data to be validated, as well as metadata used to tag which user interface control allows the user to understand and fix the error would be the most flexible for re-architecting these types of validations.

Globalization/Localization

Support for multiple languages and regional settings are an important concern for production software development produced for international markets as well as for high traffic web sites that require the ability to reach large audiences. Enterprise applications that service global companies and suppliers are no different in their globalization requirements for allowing collaboration between multiple regional user communities. These concerns are difficult to revisit once a project has been underway, and generally the globalization scope of the project should be evaluated during the initial design-time. Silverlight makes good use of the built-in provisions for globalization as part of the .NET framework, implementing satellite assemblies to store internationalized resources, as well as allowing dates and other regional settings to be formatted properly based on the CurrentCulture and CurrentUICulture as it done consistently in the .NET framework.

The main problem with Silverlight's globalization and localization settings is where it accesses the culture information from. ASP.NET bases its culture settings from information passed in the request header from the browser accessing the page, while Silverlight accesses its culture information directly from the operating system in which it's running. To avoid any possible conflicts between Silverlight page controls and other ASP.NET controls rendered on the page, it would be best to ensure that they are using the same culture information. The Silverlight wrapper class shown in the 'Handling Post Backs' section takes this into account and forces the Silverlight control to use the same culture information as the ASP.NET by priming the Silverlight host wrapper with the CurrentCulture and CurrentUICulture from the ASP.NET host page.

Handling Post Backs

Perhaps the greatest benefit of a Silverlight application is to allow storage of state information on the client machine, reducing the need to transmit information between client and server. In existing ASP.NET applications, post-backs and navigating between pages are the accepted way of updating state information, but each of these actions require completely flushing all of the client side state information stored in the browser. Allowing Silverlight components to stay resident and preserve their application state during ASP.NET post backs can be quite a problem. An extremely simple approach would be to use multiple AJAX UpdatePanels to wrap standard ASP.NET content on the page so that partial-page updates occurred asynchronously when post backs are required instead of a complete page refresh. UpdatePanel and the AJAX script management framework used to make it function can be a very quirky to use with existing code. The biggest drawbacks of this approach are that UpdatePanels do not work with ASP.NET validation as well as to a lesser extent, code errors are hidden by the UpdatePanel's asynchronous JavaScript manipulation of the page data.

A second way to handle post-backs is to host the Silverlight panel in a separate page which hosts the ASP.NET page content in an IFrame. This would enable the Silverlight control to stay resident on the page and allow the ASP.NET page to post-back and refresh as needed without affecting the parent page that contained the Silverlight content. This creates a simple low impact solution that wouldn't require rewriting how the original page operated in order to integrate Silverlight. The most obvious drawback

to this solution is the Silverlight control's integration with the ASP.NET page would be severed due to the security constraints of IFrames. Secondly, Silverlight would not be able to participate in the flow layout of the ASP.NET page; this means that the only way to force the Silverlight control to interact with the ASP.NET page's layout would be via client-side scripting. This approach would be acceptable in applications where the Silverlight control is being used as a toolbar or task pane, but would be difficult to integrate when it was required as a media element on the page or as an inline control.

A third approach is to develop a framework that can store the current state of Silverlight components, serialize the state and return it back to the server on a post back, allow server-side components to access the state information, and on re-rendering the page, de-serialize the state and return it to the Silverlight component. This method allows simple direct access between ASP.NET page components and Silverlight without changing the way that existing ASP.NET controls work. It is generally slower than the other two methods as it requires serialization of state data and the Silverlight components need to reload after a post back, but it provides for a robust method of compartmentalizing the Silverlight changes. The following code shows a simple implementation of this model.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Windows.Browser;
using System.Xml.Serialization;

namespace SilverlightTestApp
{
    [ScriptableType]
    public class ViewStateManagerSubmitShim
    {
        public event EventHandler Submit;

        [ScriptableMember]
        public void OnSubmit()
        {
            if (Submit != null)
            {
                Submit(this, new EventArgs());
            }
        }
    }

    public class KeyValuePair
    {
        public string Key { get; set; }
        public string Value { get; set; }
    }

    public static class ViewStateManager
    {
        private static Dictionary<string, string> _viewState;

        static ViewStateManager()
        {
            var shim = new ViewStateManagerSubmitShim();
            HtmlPage.RegisterScriptableObject("ViewStateManagerSubmitShim", shim);
            shim.Submit += new EventHandler(Dehydrate);

            _viewState = Hydrate();
        }

        private static void Dehydrate(object sender, EventArgs e)
        {
            var list = new List<KeyValuePair>();

            foreach (var kvp in _viewState)
            {
                list.Add(new KeyValuePair() { Key = kvp.Key, Value = kvp.Value });
            }

            ViewStateManagerHiddenField = serializeObject(list);
        }

        private static Dictionary<string, string> Hydrate()
        {
            var viewString = ViewStateManagerHiddenField;
            if (!string.IsNullOrEmpty(viewString))
            {
                var list = deserializeObject(typeof(List<KeyValuePair>), viewString) as List<KeyValuePair>;
            }
        }
    }
}
```


The mechanics of this design are fairly straightforward: a hidden field is added to the page which is used to contain and pass through the HTML encoded version of the data stored by the Silverlight control. The Silverlight control then hooks the submit event of the page, and serializes its initialization data to the hidden field, and upon reloading the control, it checks the hidden field to see if it contains initialization data and loads accordingly.

```
protected void Page_Load(object sender, EventArgs e)
{
    Page.RegisterOnSubmitStatement("PostbackSilverlightSubmit",
        "document.getElementById('Xaml1').Content.ViewStateManagerSubmitShim.OnSubmit();");
}
<asp:HiddenField ID="hdnViewStateManagerHiddenField" runat="server" Value=""/>
```

The hidden field and code to hook up the submit event to the framework needs to be added to the page. These items could be abstracted into a separate ASP.NET control which also provides provisions for accessing the values on the server side. Additionally, this separate control should use a unique key to manage its separation from any other instance of this framework on the page.

Conclusion

Silverlight has the ability to transform static ASP.NET applications into dynamic client-side applications. It provides the tools to create a rich user interface which increases the productivity of users. This translates into an increased return on investment for clients as well as decreased development time for future enhancements due to improved software architecture and the built-in division of work. This article shows how existing ASP.NET applications can be iteratively restructured into Silverlight based applications. This approach allows clients to redesign their solutions as part of a maintenance cycle, reducing both cost and the risk of scope creep.

Deploying discrete Silverlight controls upfront, as part of the first few iterative cycles helps clients to see a high degree of short-term improvement and their feedback on these controls helps to shape future requirements. Silverlight allows creativity to be the only boundary to developing solutions to fit client needs, including needs they never knew existed with legacy ASP.NET applications. Long term benefits of integrating Silverlight include the default separation of concerns brought about from the tiered service oriented architecture as well as the ease by which designers and developers can collaborate on user interface designs. Clients wishing to re-architect solutions while still maintaining a high level of availability and flexibility for change will find Silverlight a great fit for their development needs.

About Risetime:

Since 1984, Risetime has been serving the needs of organizations looking for a strong partner to provide business and technology solutions and services. Headquartered in Chicago and a Microsoft Gold Certified Partner, Risetime focuses on providing a wide range of IT consulting services as well as business process management, content management, and web solutions.

547 West Jackson Blvd. 8th Floor
Chicago, IL 60661
P 312.362.9930
F 312. 362.9925

info@risetime.com



www.risetime.com