Building an InfoScreen Solution with Silverlight 1.0

Maximilian Knor,
<max.knor@microsoft.com>
Mario Szpuszta
<marioszp@microsoft.com>

This paper shows implementation concepts for building a dynamic InfoScreen application with Silverlight. The ideas and concepts demonstrated in this paper are based on a prototype Microsoft Austria has built together with derStandard.at – one of Austria's largest web sites and news papers. derStandard.at is currently developing a solution based on Silverlight that will be hosted on their web site and on InfoScreen panels in public locations. The application displays different kinds of information such as news, weather forecasts, stock trade information, sports information which is being downloaded dynamically from backend REST-style services including the information itself as well as styles and animations.

Microsoft Austria

derStandard.at

6/4/2008

Table of Contents

Introducing the Project and its Requirements

derStandard.at is one of the largest web sites in Austria, attached to one of the most important newspapers of the country. Compared to other news sites, derStandard.at is seen as some kind of a country wide thought leader. Therefore many people, especially intellectual influencers, are actively posting in their online forums, discussing a variety of topics reaching from economics to IT and technology. That is one of the main reasons why derStandard.at wants to demonstrate technology leadership and innovation on the market and also a main driver for moving towards Silverlight.

derStandard.at is currently evaluating new ways of presenting information in addition to the well known, old fashioned static way of presenting content on typical news paper web pages. While their web site has been migrated to ASP.NET and ASP.NET AJAX extensions this winter they started working on a new, innovative Silverlight project in parallel. This project is about presenting several kinds of information in an interactive and modern way through a variety of channels. derStandard.at wants to achieve this goal by building a platform for a special breed of applications which can be hosted on their web site as well as on dedicated flat-screen panels in public locations such as railway stations, casinos or airports. In any case these applications are downloading information and the methods and styles for presenting the same (such as animations, images, videos etc.) from backend systems through REST-style web services. They call these breeds of applications InfoScreen apps.

We supported derStandard.at with an architectural design and the development of a prototype for a smooth start of the development process for this new breed of applications. In this paper we will describe the experience and approaches for designing and implementing such types of applications with Silverlight 1.0 which we have worked out together with derStandard.at. Currently derStandard.at is working on a Silverlight 2.0 based solutions. But as Silverlight 1.0 is released, already, we decided to write this paper on the original prototype built which we've built with Silverlight 1.0 as well. We will maybe update the paper at a later point of time to Silverlight 2.0 if anyone is interested in seeing such an update for this paper. In any case the concepts presented in this paper apply to both versions of Silverlight in a very similar fashion.

Understanding the overall Architectural Approach

During architecture design-sessions derStandard.at challenged us with several key requirements the new application platform needed to adhere to. The first and foremost was that the platform needs to fit into their current architectural approaches for delivering content to the browser. Essentially all of the content is managed through a custom content management system. The ASP.NET front-end layer gets the content as XML and transforms it via XSL/T style sheets into the rendered content before it gets delivered to the browser. Second the architecture for the new platform needs to be as lightweight as possible which means that usage of a heavy-weight infrastructure such as full-blown SOAP-based web services was not really an option.

Taking the first requirement of getting content from a CMS formatted as XML we came up with the idea of creating special XSL/T style sheets for the final implementation of the application that transforms content to XAML which is dynamically loaded into a Silverlight "container" application at runtime. The second requirement resulted in the decision of using WCF with REST-style channels configured so that information can be queried in a lightweight style through plain XML HTTP GET/POST requests as well as JSON-formatted requests and responses.

In this paper we will focus on the design and implementation of the Silverlight front-end and in the examples we will use simplified back-end systems to mimic the behavior of the actual back ends of derStandard.at.

Let's start with a look at the overall design of the Silverlight application. As we mentioned earlier, the most recent content delivered as XAML should be downloaded continuously into a Silverlight container application that runs on the client. So the first step is taking a look at this container application. Essentially we defined several content regions with derStandard.at for this container application. One region is used for rendering the main content while the other regions are used for displaying static types of content updated frequently such as weather forecasts, stock tickers or similar types of information. The following figure demonstrates that layout of the container application:

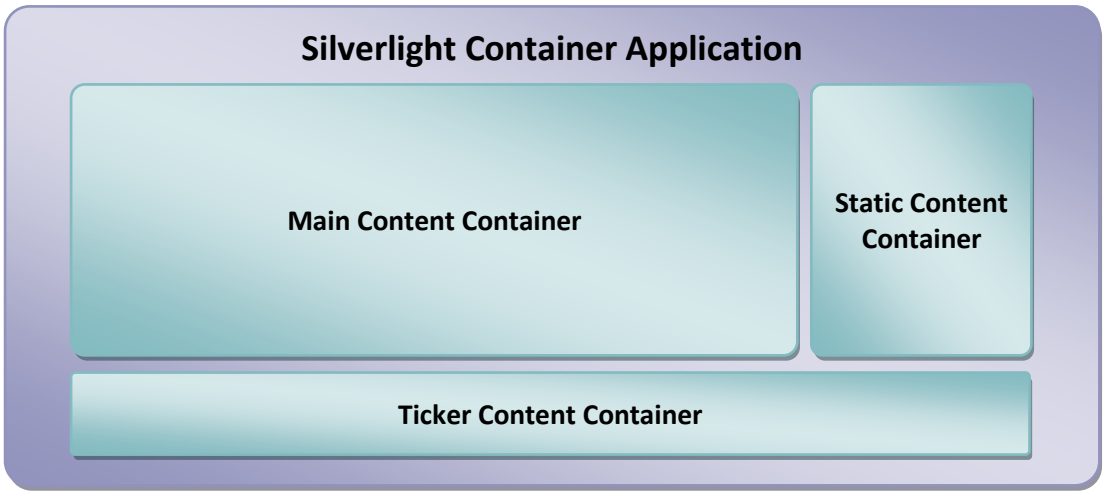

Figure 1: The Silverlight Container Application Design

The main-content container displays rather complex content that includes the actual information as well as animations, images, videos and rich styles for presenting the information. This content is downloaded as pre-built XAML from the backend services so that the container does not need to bother about building the actual visual tree with layout specifics. The ticker container and the static content container rather contain static XAML that only downloads specific information such as images and text from the services in regular intervals. Downloading pre-built XAML for the main content panel and rather simple information for the smaller static content panels brought us to the idea of splitting the backend services into two kinds of services:

- ✓ **Layout services** which are providing ready-to-use XAML content that is created on the server via XSL/T transformations out of the content from the CMS. This content is created by the editorial staff of the online news paper.
- ✓ **Data Services** on the other hand are just delivering data that is filled into the static panels of the container application. Therefore the animations and the style of the presentation is not delivered through these channels and is pre-defined in the Silverlight container application.

Now that we have identified all the necessary actors and their responsibilities we can go one step further with our thoughts. The next step is thinking about the actions the application and these actors will complete during runtime and in which order they will complete these actions – we call

that the life cycle of the InfoScreen application. The following figure illustrates the life cycle of the application:

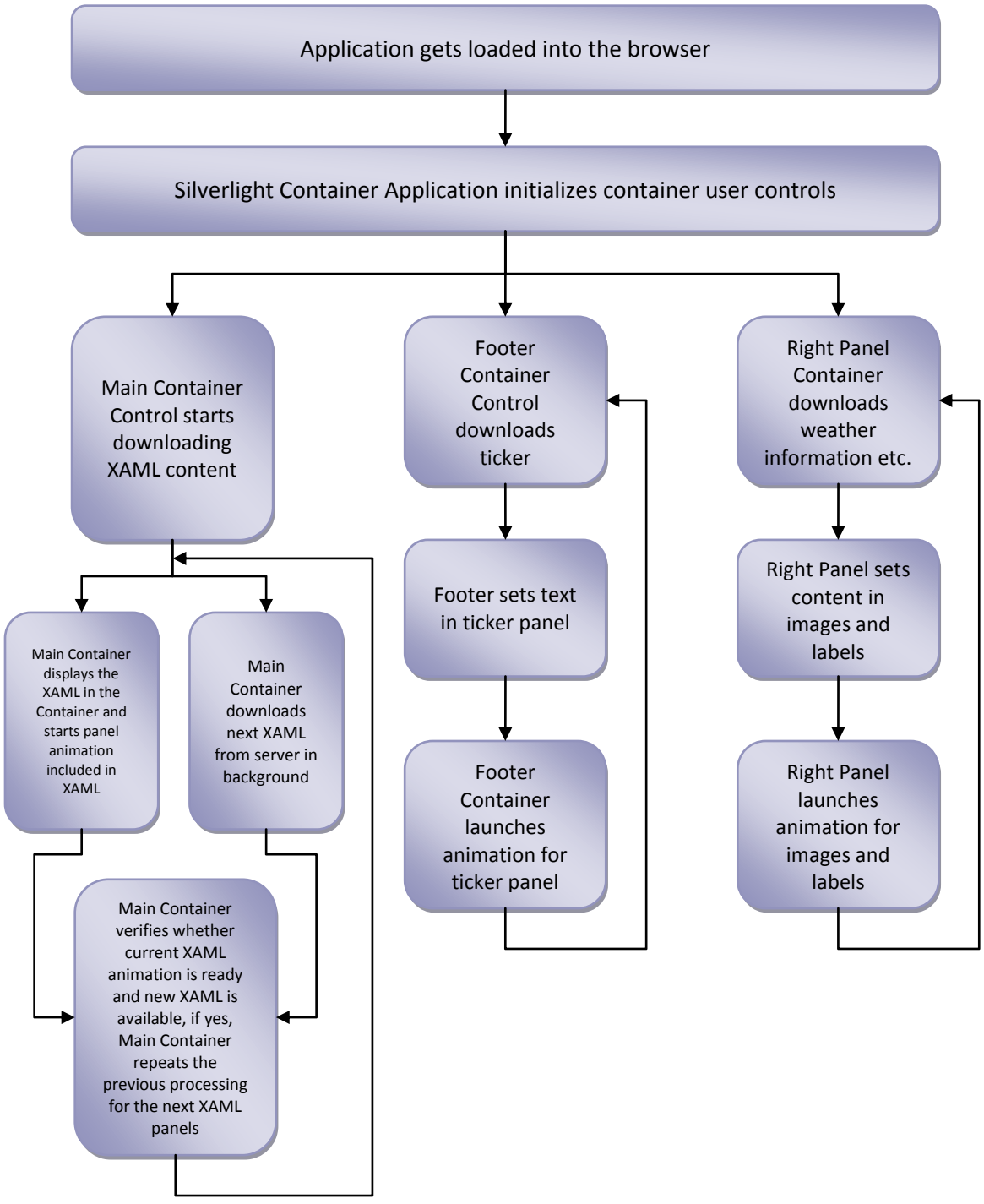

Figure 2: The Life Cycle of the Silverlight InfoScreen Application

There are two interesting points with this life cycle. The first one is Silverlight-related; the second one is related to the processing of the animated content panels. First of all the life-cycle you see in the previous figure runs continuously as long as the Silverlight application is displayed in the browser after a single request. The life-cycle stops as soon as the user starts a browser navigation action that leaves, submits or refreshes the page. That means, for example, when the user hits the F5 button for refreshing the page, the life-cycle of the Silverlight application stops, a GET request is submitted to the server, the Silverlight application is downloaded (either from the server or the browser cache if nothing changed) and restarted.

More interesting is the second point which is related to the timing of the animation runtime of a downloaded XAML panel and the time necessary for downloading the next XAML. As you can see in the previous figure we have to deal with two specific situations here:

- ✓ The animation of the downloaded XAML panel runs longer than the download of the next XAML panel. In that case immediately after the animation from the previous XAML panel has finished we can display the next XAML panel (downloaded, already) and start its animation.
- ✓ If the animation of the currently displayed XAML panel is shorter as the required time for downloading the next XAML panel, the currently displayed XAML panel needs to stay in the container as soon as the download is completed.

Therefore the decision tree displayed in the previous figure needs to be reused within two event handling callbacks in the application as you will see later in detail: in the callback that gets called when the XAML animation of the currently displayed XAML panel is completed and in the callback that gets executed when a download of a XAML panel is completed.

With that we have covered all the necessary details for building the InfoScreen application platform from an architectural point-of-view. Therefore in the next section we will discuss parts of the implementation details of the prototype we have implemented together with derStandard.at.

Building the Silverlight Application

Essentially we have split the development teams into two parts for building the InfoScreen prototype: one team was responsible for building REST-style backend services based on WCF that provide the previously mentioned Layout- and Data-Services. The second team was responsible for designing the Silverlight application, its controls and its logic elements.

Preparing the REST-Style Services

Before we actually started splitting the team into two parts we design the contracts for the REST-style web services. Due to the fact that we used JSON serialization a contract first approach really helped us with the team-split because both teams were able to rely on the object structures we defined in the service interfaces for the REST-style web services – one team just leveraged them from within JavaScript, the other team leveraged them as .NET Classes (WCF Data Contracts) in the .NET backend service implementations. Simple first versions of the contracts we used for the Content Service in the prototype that returned dynamically generated XAML looked as follows:

```
[ServiceContract(Namespace="derStandard.at/prototype")]
public interface IContentService
{
    [OperationContract]
    [WebGet(ResponseFormat=WebMessageFormat.Json)]
    ContentResponse GetContent(string contentSection,
                               string sectionStateInfo);
}
```

The important part on these service interface definitions is the usage of the [WebGet] attribute where we defined that the response of the GetContent() method where we defined that the response is returned in a JSON serialized format. This format allows JavaScript code to create JavaScript objects directly out of the response of the WCF service. One additional think you have to bare in mind when using the [WebGet] attribute is that you also need to enable REST-support in the WCF service model configuration of your web.config file in you web site project for the service as follows:

```
<system.serviceModel>
  <behaviors>
    <endpointBehaviors>
      <behavior name="ContentServiceAspNetAjaxBehavior">
        <enableWebScript />
      </behavior>
    </endpointBehaviors>
  </behaviors>
  <serviceHostingEnvironment aspNetCompatibilityEnabled="true" />
  <services>
    <service name="StandardContentService.ContentService">
      <endpoint address=""
                        behaviorConfiguration="ContentServiceAspNetAjaxBehavior"
                        binding="webHttpBinding"
                        contract="StandardContentService.IContentService" />
    </service>
  </services>
</system.serviceModel>
```

The response included several attributes that have been evaluated by the Silverlight client application in JavaScript code. We have encapsulated these into the ContentResponse class as follows:

```
[DataContract(Namespace="derStandard.at/prototype")]
public class ContentResponse
{
    [DataMember]
    public string State { get; set; }
    [DataMember]
    // Storyboard for the template is called "ContentStory"
    public string Content { get; set; }
}
```

The State-property included a current ID for the downloaded XAML content. As you can see in the interface definition of the IContentService, this state is passed into the GetContent() method as the sectionStateInfo parameter along with the region of the Silverlight client application. The state allowed the backend service to decide which content is the next one to be returned for the region passed in through the contentSection parameter in this interface. This interface design makes it possible to reuse the content service if later versions of the InfoScreen panel will have more than one content container displaying rich content generated and returned from the server as XAML.

As the implementation of the services is out of scope for this whitepaper we move on to the implementation of the actual Silverlight front-end. All that you need to know is that the Content property of an instance of the ContentResponse class returned by the service contains XAML. This XAML is generated by merging XML content returned from the internal CMS system with XAML-templates stored as XML files in a database by using XSL/T transformations. If you want to re-implement the concepts we recommend starting with static XAML content that you just return through the defined service interfaces.

Creating the Silverlight "Shell-Application"

After we designed the contracts we split into two teams – one team was implementing the services described in the previous section while the second team has been working on the Silverlight frontend. And that's exactly the one we are explaining here. As mentioned earlier we have used Silverlight 1.0 because it was available at the point of time when we started the prototyping. For more information on how to create a new project for Silverlight applications take a look at the Silverlight 1.0 SDK available on the Microsoft Downloads Web Site.

Using ASP.NET AJAX for backend-access

Before we actually start we need to talk about the communication to the backend services. As you have seen in the previous section we used Windows Communication Foundation (WCF) and the [WebGet] attribute to use JSON as response format for being able to return content from the Web Service in a way that can be understood by JavaScript, easily. JavaScript actually is able to take the response from the service and create JavaScript objects in code without any further implementation steps a developer needs to perform.

Finally it leaves one question open – how can we call this web service without too much effort. And the answer was the usage of the ASP.NET AJAX extensions. When using the ASP.NET <asp:ScriptManager> server control it generates JavaScript code that allows us to call the web services automatically. All you need to do is adding a service reference as part of the < asp:ScriptManager> server control tag as follows:

```
<asp:ScriptManager ID="SM1" runat="server">
    <Services>
        <asp:ServiceReference Path="~/ContentService.svc" />
    </Services>
    <Scripts>
        <asp:ScriptReference Path="~/Silverlight.js" />
        <asp:ScriptReference Path="~/Default_html.js" />
    </Scripts>
</asp:ScriptManager>
```

The <asp:ServiceReference> tells the <asp:ScriptManager> server control that it should take the contract definition of the ContentService.svc WCF-based service to generate a JavaScript proxy. Also notice that we are including other JavaScript files independent from ASP.NET AJAX through the script manager, as well. We split the JavaScript code required for the Silverlight client into several .JS files for maintainability and readability. The Silverlight.js script file contains Silverlight specific stuff while the Default_html.js contains JavaScript code for our implementation. We've included them all through the <asp:ScriptManager> as when using ASP.NET AJAX the <asp:ScriptManager> takes on the role of managing all JavaScript code rendered to the client – whether it is AJAX related or not. As such it replaces the standard ASP.NET Page.ClientScript manager instance which is responsible for JavaScript in ASP.NET applications without the usage of ASP.NET AJAX.

The basic Layout of the application

Essentially the layout is all about putting together a number of Canvas controls. The Canvas is, unfortunately, the only layout control included with Silverlight 1.0 – so that's where we see room for improvement for future versions built with Silverlight 2.0. Anyway – we use the Canvases as a mechanism for grouping content on our overall screen. The main Page.xaml for our Silverlight application therefore looks as simple as follows:

```
<Canvas
      xmlns="http://schemas.microsoft.com/client/2007"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Width="1280" Height="720"
      Background="Blue"
      Name="MainPage"
      >
      <Canvas Background="Red" Width="932" Height="591"
            Canvas.Left="8" Canvas.Top="8" Name="MainContainer"/>
      <Canvas Background="Green" Width="327" Height="591"
            Canvas.Left="944.5" Canvas.Top="8" Name="RightContainer"/>
      <Canvas Background="Yellow" Width="1264" Height="116"
            Canvas.Left="8" Canvas.Top="603.5" Name="BottomContainer"/>
</Canvas>
```

When loading content from the server, we do nothing else than taking the XAML returned from the server as a string, convert it into a XAML visual tree and add it as a child to one of the container canvas controls of our main Page.xaml. For the transitions between contents every XAML downloaded from the server needs to have one root element of type Canvas. This root canvas of the returned XAML in turn contains all animations for presenting the content including fade-in and fade-out animations which can be used to "simulate" transitions between content-pages. A sample XAML returned by the server therefore can look as follows:

```
<Canvas
      xmlns="http://schemas.microsoft.com/client/2007"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Width="932" Height="591" Opacity="0" x:Name="canvas">
      <Canvas.Resources>
            <Storyboard x:Name="Animate">
                  ... Animation Stuff goes here ...
            </Storyboard>
      </Canvas.Resources>
      <TextBlock Width="916" Canvas.Left="8" Canvas.Top="8"
                 Text="Sample Title"
                 TextWrapping="Wrap" FontSize="36"
                 x:Name="TitleTextBlock"/>
      <Image Width="300" Height="200" Canvas.Left="324" Canvas.Top="94"
             x:Name="TeaserImage" Source="/Images/Test.jpg"/>
      <TextBlock Width="916" Height="263" Canvas.Left="8" Canvas.Top="320"
                TextWrapping="Wrap" FontSize="20"
                x:Name="LeadTitleTextBlock">
            This is the content of your XAML page...
      </TextBlock>
</Canvas>
```

Now the next question is which code we need to dynamically display and download such XAML fragments from the server. We address these in the subsequent two sections.

Content Container User Controls, Download and Transition Implementation

When doing the implementation of the three content containers, we looked for a way of how to encapsulate their logic and keep it together with the corresponding UI. JavaScript helps us by providing the concept of prototype classes, a method to simulate object oriented behavior in JavaScript. What we did is implementing a separate class for each of the content containers (e.g. the main content container of our Page.xaml main Silverlight application) that encapsulates the download and transition logic within the class. For those of you not familiar with JavaScript prototypes, here's how you would implement a simple class in JS first:

```
InfoWeb.MainContentLoader = function()
{
}

InfoWeb.MainContentLoader.prototype =
{
      initialize: function(control, parentElement)
      {
      // method logic
      }
}
```

With the combination of XAML as UI and JavaScript prototype classes we have a concept very similar to user controls in other environments such as WPF. Therefore we named our content containers "user controls". Basically what the JavaScript prototype class does is manipulating parts of the XAML it will be associated with such as its corresponding canvas (e.g. the main content canvas), adding children to it or starting animations of the children, etc.

In our case we are building a user control for the main content container that is responsible for displaying XAML downloaded from the previously created WCF service at runtime. Of course every user control needs to have the capability to initialize itself. That can be done through the JavaScript initialize() prototype method. As JavaScript prototype classes are not automatically connected to their XAML-UI we need to wire up the class with the XAML visualization by ourselves in the initialize method. In our case that's the Canvas defined for the main content container in our Page.xaml before (the Canvas with the name "MainContainer"). To wire up the JavaScript prototype class with its XAML we pass in the parentElement parameter in the sample above while the second parameter, control, is just a reference to the Silverlight hosting control. We need a reference to the Silverlight hosting control to be able to call functions on the Silverlight plug-in as well. After wiring up with the UI the control starts downloading the first content part out of initialize as follows:

```
InfoWeb.MainContentLoader.prototype =
{
      initialize: function(control, parentElement)
      {
          this.animations = {};

          // Capture the current ID / state of the displayed content
          this._id = "-1";

          // Reference to Silverlight hosting control
          this._control = control;
```

```
            // Reference to owned XAML tree (in our case the MainContent
            // Canvas of Page.xaml)
            this._parentElement = parentElement;

            // Reference to XAML tree of the current content
            this._currentContent  = null;
            // Reference to the XAML tree of the next content
            this._upcomingContent = null;

            // State of current download
            this._currentContentFinished = false;

            this.getData();
        },
        getData:function() {
            var service = new derStandard.at.IContentService();
            service.GetContent("MainSection", this._id,
                this.getContent_Succeeded, this.getContent_Failed, this);
        },
        // ... more JavaScript Code omitted
}
```

As you can see above the getData() method uses the JavaScript web service proxy generated by the <asp:ScriptManager> object for our WCF based web service. The first two parameters are the parameters of our WCF service method (contentSection and sectionStateInfo whereas the state-info is encapsulated in the _id member of our prototype class). The third parameter is a reference to a JavaScript method of our prototype class that is called by the service proxy when a download completed successfully while the fourth parameter is a reference to a JavaScript method that is called when the request to the server failed for some reason. With the last parameter we are able to keep context information for the asynchronous call that allows us to call back on the UI thread in our case by passing a reference to the JavaScript prototype class instance itself. Next we will take a look at the callback method that gets called when the download of the XAML from the server succeeded:

```
    // ... previous JavaScript code omitted
    getContent_Succeeded:function(result, context) {
        var xamlFragment = result.Content;
        context._id = result.State;

          context._upcomingContent =
              context._control.content.createFromXaml(xamlFragment);

        if (context._currentContent == null ||
            context._currentContentFinished)
        {
            // Initial Load
            context.switchContent();
        }
    },
    getContent_Failed:function(result, context) {
        alert("Error: " + result);
    },
    // ... more JavaScript code omitted ...
```

The method gets called by the ASP.NET AJAX infrastructure after the asynchronous JavaScript call to the web service is completed. As parameters it receives the result of our web service and the context parameter (which is a reference to the control as we passed in this as last parameter in the web service call within the previous code sample) as parameters. The result is a JavaScript object de-serialized out of the JSON response from the WCF web service. This result contains the state (which corresponds to the ID of the current panel of the InfoScreen application) as well as the XAML fragment as strings (see ContentResponse class of our WCF contract earlier).

Therefore we take the XAML string and call the createFromXaml() method of the Silverlight plug-in to create a visual tree out of the string. Next we check whether the currently displayed XAML has finished its work, already, and if so we transition to the new content by calling the switchContent() method of our control. You can see the switchContent() method in the following code snipped:

```
        // ... previous JavaScript Code omitted ...
        switchContent:function()
        {
          // Switch
          this._currentContent = this._upcomingContent;
          this._upcomingContent = null;
          this._currentContentFinished = false;

          this.animations.contentStory =
                  this._currentContent.findName("Animate");

          this.animations.contentStory.addEventListener("Completed",
                Silverlight.createDelegate(this,
                                           this.contentStory_Completed));

          this._parentElement.Children.clear();
          this._parentElement.Children.add(this._currentContent);

          this.startAnimation();
          this.getData();
        },
        startAnimation:function()
        {
            this.animations.contentStory.stop();
            this.animations.contentStory.begin();
        },
        contentStory_Completed:function(sender, e)
        {
            // Is there a new ContentStory already loaded?!
            if (this._upcomingContent != null)
            {
                // Switch to new one!
                this.switchContent();
            }
            // Else stick with old one!
            else{
                this._currentContentFinished = true;
            }
        }
}
```

In the switchContent() method we start with updating references to the current and the upcoming XAML content displayed in the main content container. As we parsed them into a XAML visual tree using the createFromXaml() method in the getContent_Succeeded() function earlier, this is a full-blown object tree with methods on it. Therefore we can retrieve the animation from the downloaded XAML by using the findName() method on the _currentContent member which we've set to the new XAML visual tree in the lines of code before. We keep the animation in the local animations member to be able to use it on other functions of our prototype such as the startAnimation() method, later. As we need to know when the animation has done its work we subscribe to the Completed event by using the addEventListner() method of the retrieved animation. Afterwards we clear the children currently displayed in the main content container which is kept in the _parentElement member of the JavaScript prototype class. After we have deleted the old content from the main content container we add the new XAML visual tree to the main content container. Finally we start the animation by calling the startAnimation() method and we download the next content by calling the getData() method, again.

As you can see, the startAnimation() method of our JavaScript prototype is pretty simple. It just takes the animation we have stored into the animations-member of our prototype class and calls the begin() method to start the new animation. Finally we have the contentStory_Completed event routine which is called whenever an animation has done its job. In this method we just verify, whether the new content is available, already, by check the _upcomingContent member of our class. If yes, we switch to the upcoming content by calling the switchContent() method again. If not we stay with the old one and set the _currentContentFinished flag to true to indicate, that the next content can be displayed as soon as the download has been finished (look to the implementation of the getContent_Succeeded() function for more details).

Finally that's it. We are done and we have implemented a Silverlight InfoScreen application that downloads XAML at runtime from a REST-style web service, displays it in a pre-defined content region and starts animations to have smooth transitions as well as other cool stuff for presenting information actively working in the browser.

Conclusion, Summary

In this article you have learned about capabilities of Silverlight 1.0 for building applications that run in the browser, update content frequently by continuously download information from a backend and presenting that information with smooth animations. Also you have seen how Silverlight 1.0 works together with the ASP.NET AJAX extensions very well due to the comfortable capabilities of the ASP.NET AJAX extensions for calling backend web services.

The concepts you have learned about in this paper have been developed within a prototyping engagement together with derStandard.at which is currently working on a Silverlight-based platform for implementing InfoScreen applications hosted on their web and on flat panels in several public locations. Right now derStandard.at is investigating in Silverlight 2.0 for future work on the InfoScreen platform. Of course we will update the contents of the paper to Silverlight 2.0 if anyone is interested into such an update.

Finally we hope that you got something useful out of the concepts, approaches and implementation details we've described in this paper and we are more than happy about any feedback!