



Building a Multi-Page Image Viewer with ImageGear for Silverlight

As the desire to deliver rich web content and functionality has increased over the last decade, Rich Internet Application (RIA) technologies have become increasingly powerful. Unfortunately, many of these technologies have been out of reach for many client application developers because of the non-trivial learning curve involved. In addition, with the number of RIA technologies available, where do you even begin? In many cases, there was not a “one size fits all” solution, so RIA development would involve a mix of HTML, JavaScript, Adobe Flash, and perhaps a little AJAX thrown in (just to name a few possibilities). Luckily, Microsoft has entered the world of RIA frameworks, and with the introduction of Microsoft Silverlight 2, offers a platform, which leverages existing .NET developers’ talents. With version 3.0 in the works, providing enhanced graphics support, data binding, and perhaps best of all, out-of-browser support, the Silverlight platform is certainly worth a look for new, and even existing, RIA development.

Silverlight includes many of the same base services and types included in the .NET Framework. However, because it is a runtime built specifically for the web, where developers expect a robust platform in a small package, much of the functionality included in its desktop cousin is absent. For example, the image type in Silverlight, **System.Windows.Media.Imaging.BitmapImage**, only supports JPEG and PNG image file types, and does not include support for grayscale. This is where third-party tools vendors become part of the Silverlight ecosystem, providing extensions to the base platform as the market demands. For the imaging domain, [Accusoft](#) has continued its reputation as the imaging tools leader, by providing one of the first imaging toolkits for Silverlight developers – ImageGear for Silverlight.

This article will provide a quick tour of the [ImageGear for Silverlight](#) toolkit while developing a multi-page image viewer, running completely on the client, via managed code.

Building the Viewer

To get started, download the ImageGear for Silverlight SDK and sample code [here](#). Open Visual Studio 2008, and bring up the **New Project** dialog. Under the **Visual C#** heading, select **Silverlight** as the project type, and then select **Silverlight Application** as the project template. If you do not have **Silverlight** as an available project type, you need to install the Silverlight 2 SDK, available from Microsoft. Finally, enter a name for your project, and click OK (Figure 1).

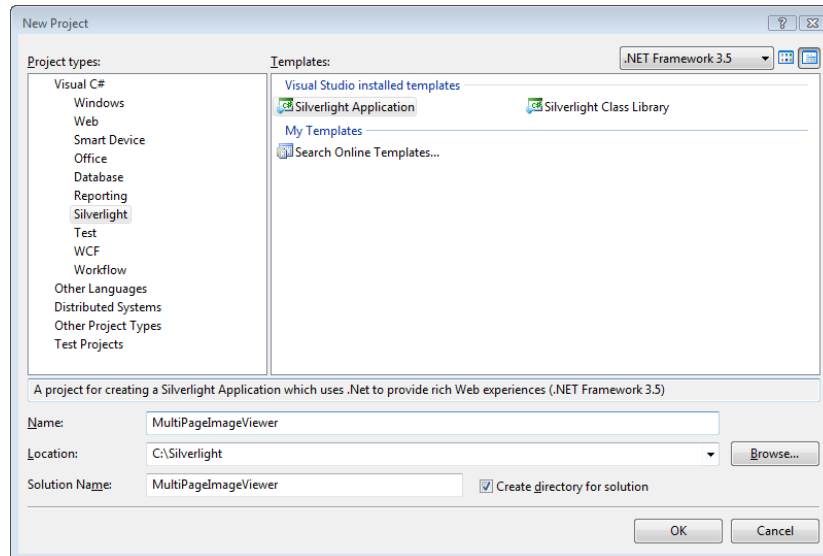


Figure 1: Defining the Silverlight Project

The next screen presented as you create the project deserves a bit of explanation, particularly to those new to Silverlight development. The unit of deployment of a Silverlight application is a XAP package, and deploys within an HTML web page, or as part of an ASP.NET project. For many Silverlight applications, using an HTML web page as your host would be perfectly fine. However, ImageGear for Silverlight utilizes a web service to validate its SDK license, and the Silverlight application calling that web service must run from the domain name that hosts the service. So, it is best to avoid the HTML web page option, and instead create the ASP.NET project. The project wizards set everything up for you anyway (Figure 2), so even the most inexperienced web developer will be ok.

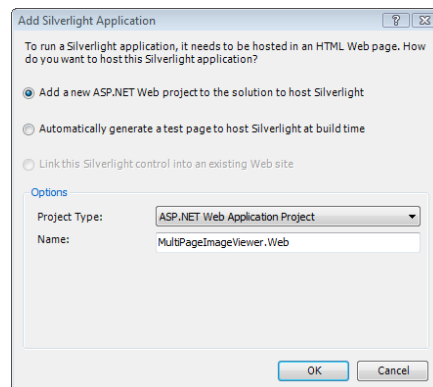


Figure 2: Selecting the Silverlight Host

And, with a click of the OK button, a solution is created, containing two projects – the Silverlight application, where we will spend most of our time, and an ASP.NET host, set as the startup project. If you compile and run at this point, the application will execute; however, it will be pretty boring – just a blank page in IE. However, this page is not blank. There is a full-fledged



Silverlight application running within it. We just need to build its layout to support our image viewer.

Defining the Layout

If you have utilized **Windows Presentation Foundation (WPF)** or **Windows Workflow Foundation (WF)**, you have likely become familiar with XAML. While XAML is nothing more than a way to initialize a set of .NET types, it has become very popular in the next generation GUI stacks provided by Microsoft, and that carries through to Silverlight. To define the layout for our multi-page image viewer, we will modify the generated XAML to specify a user interface more suitable than the blank surface we get by default. A complete overview of XAML and the Silverlight GUI stack is beyond the scope of this article, but we will briefly explain what we are doing along the way.

To help us get our bearings, Figure 3 below shows the default view and XAML generated by Visual Studio. It defines a Silverlight **UserControl**, which contains an empty **Grid**.

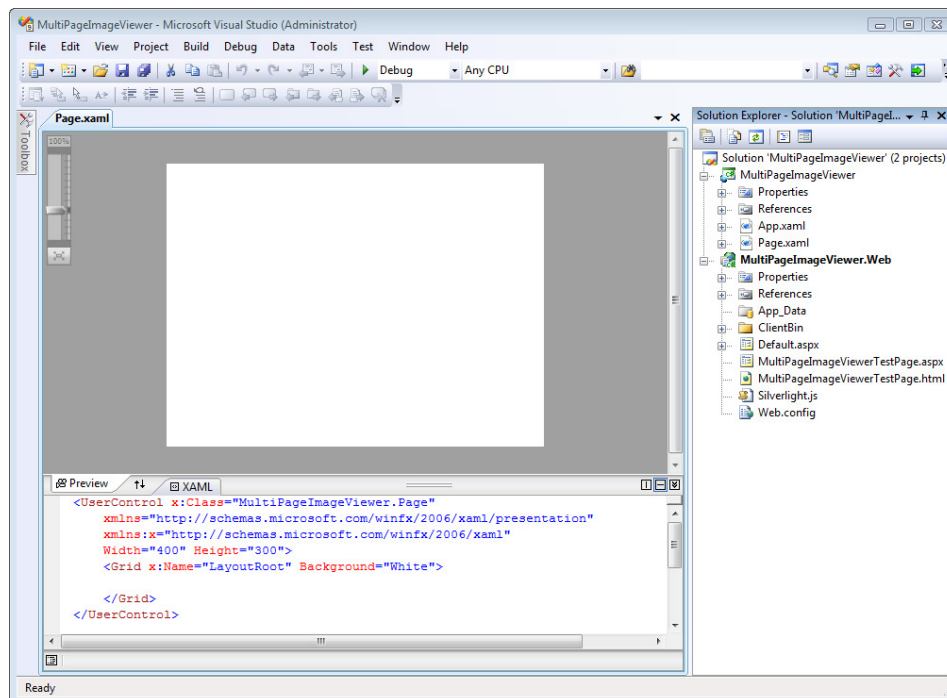


Figure 3: Generated Page.xaml

For our image viewer, we want to add a couple of controls to the **UserControl** – an instance of the **ImGearSilverlightPageView** type for display of the image and several buttons to support opening a file and navigating between its pages. But, before we modify the XAML, we have some references to add to the project to ensure calls to ImageGear for Silverlight resolve at compile time. ImageGear for Silverlight allows the developer to deploy only those assemblies required for their specific application. This allows the developer to keep the size of their deployment packages small, while enabling background loading of non-essential assemblies. A complete listing and description of the assemblies in ImageGear for Silverlight are in the toolkit's



documentation. For now, it is enough to make sure your references list looks like that in Figure 4. You will find the ImageGear for Silverlight assemblies in your \\Program Files\Accusoft\ImageGear for Silverlight\Bin directory.

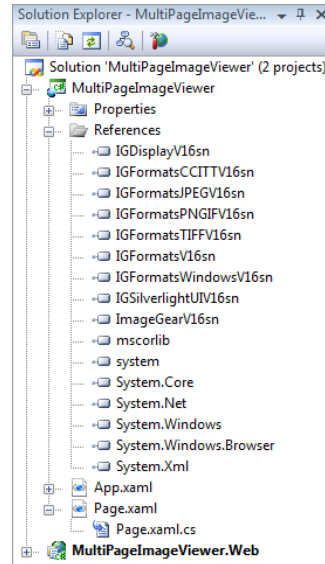


Figure 4: Required References

Finally, Listing 1 (below) shows the XAML for our image viewer's layout. The XAML in Listing 1 adds two rows in the original grid, one to hold the image viewer and a second to hold the buttons. For the **ImGearSilverlightPageView**, it is placed directly in the cell because it will simply fill its available space. The buttons, however, require some additional layout containers to ensure the desired layout is created. This is done with a second **Grid** containing two columns. The first will contain a **Button** we will use to invoke an Open File operation. The second column will contain two buttons – Previous Page and Next Page – so we will use a **StackPanel** container, and place two buttons within it. The other primary point of interest in our XAML is the use of a **Style** resource for each of our buttons. To make our XAML a bit cleaner and more efficient, the look for our buttons is defined once, as a **Style** contained in the **UserControl** resources, and applied to each button. There are many excellent resources on the Silverlight GUI elements and XAML, and if you are not familiar with these technologies, I encourage you to look. You will never want to drag and drop user controls again!



Listing 1: Multi-Page Image Viewer XAML

```
<UserControl x:Class="MultiPageImageViewer.Page"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns:IGSilverlightUI="clr-namespace:ImageGear.Silverlight.UI;assembly=IGSilverlightUIV16sn"
  MinWidth="640" MinHeight="480">

  <UserControl.Resources>
    <Style x:Key="ButtonStyle" TargetType="Button">
      <Setter Property="Width" Value="100"/>
      <Setter Property="Margin" Value="8,4,8,4"/>
    </Style>
  </UserControl.Resources>

  <Grid x:Name="LayoutRoot" Background="White">
    <Grid.RowDefinitions>
      <RowDefinition Height="9*"/>
      <RowDefinition Height="*/>
    </Grid.RowDefinitions>

    <!-- Primary Image Viewer -->
    <Grid Grid.Row="0" Background="Gray">
      <IGSilverlightUI:ImGearSilverlightPageView Margin="8"
        x:Name="mPageView" Background="Gray"/>
    </Grid>

    <!-- Button Row -->
    <Grid Grid.Row="1">
      <Grid.ColumnDefinitions>
        <ColumnDefinition Width="1*"/>
        <ColumnDefinition Width="2*"/>
      </Grid.ColumnDefinitions>

      <Button x:Name="mOpenButton" Grid.Column="0" Style="{StaticResource ButtonStyle}"
        Click="mOpenButton_Click" HorizontalAlignment="Left" IsEnabled="False">
        <TextBlock Text="Open File"/>
      </Button>

      <StackPanel Grid.Column="1" Orientation="Horizontal" HorizontalAlignment="Right">
        <Button x:Name="mPreviousButton" Style="{StaticResource ButtonStyle}"
          Click="mPreviousButton_Click" IsEnabled="False">
          <TextBlock Text="&lt;&lt; Previous"/>
        </Button>
        <Button x:Name="mNextButton" Style="{StaticResource ButtonStyle}"
          Click="mNextButton_Click" IsEnabled="False">
          <TextBlock Text="Next &gt;&gt;"/>
        </Button>
      </StackPanel>
    </Grid>
  </Grid>
</UserControl>
```

Now, it is difficult to visualize our image viewer using the XAML definition only. Luckily, Visual Studio supports a split view so you can see the result of your XAML as you go. Figure 5 below shows the layout generated by the XAML in Listing 1.

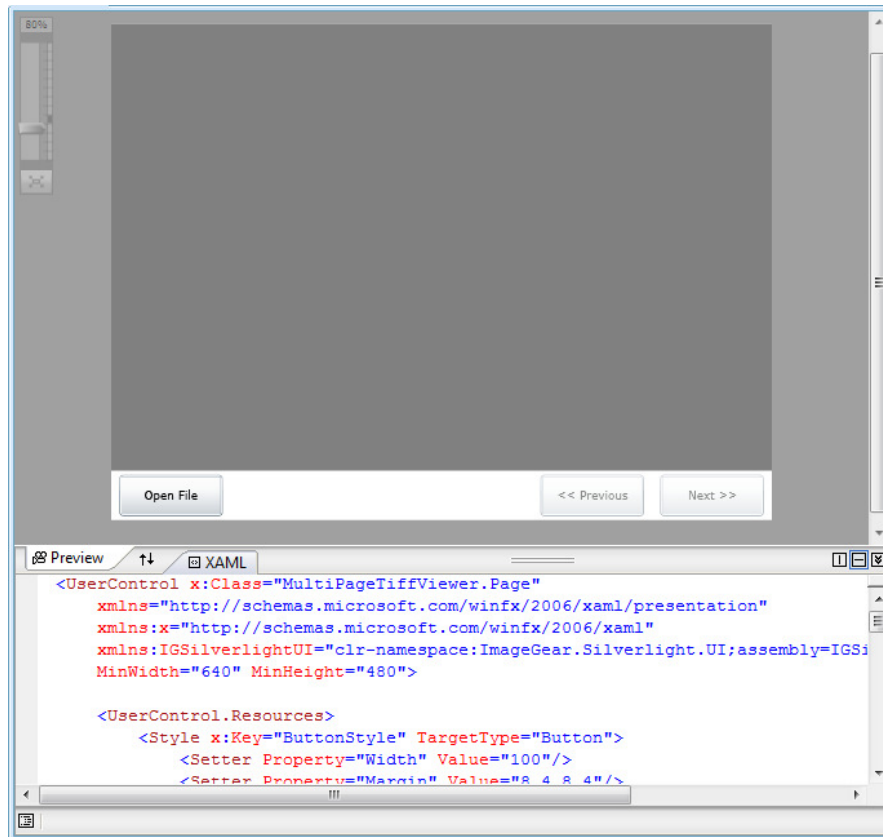


Figure 5: Multi-Page Image Viewer Layout

We have come a long way from the blank page, but now we need to write the code for our application to run. If you compile and run right now, you will face all sorts of errors (missing click handlers for one), so let's take care of those, and get our image viewer running!

The Code

We will start with the initialization code, contained in the **Page** constructor shown in Listing 2 below. ImageGear for Silverlight requires a license to run, and while you develop your application with the SDK, it uses a web service for validation.¹ The **ImageGearLicense.SetService** call allows you to specify the location of this web service, and it needs to be running from a URL on the same machine as the toolkit. Next, an anonymous delegate is setup to run once the **LicenseRequest** event fires, more on this soon. Finally, **ImageGearLicense.SetSolutionName** is called, passing the evaluation solution name as an argument.

The anonymous delegate that executes when the **LicenseRequest** event occurs is where the remainder of the initialization occurs. The event is required because of the web service involved; when the request for a license is made to the web service, the response is not synchronous. As a result, we have to wait until the request is granted before proceeding further.

¹ For more information on deployment licensing, please see the ImageGear for Silverlight documentation.



Within the delegate body, initialization of the PNG and Windows (bitmap) codecs are performed, along with CCITT TIFF, TIFF, and JPEG. While the toolkit requires the PNG and Windows formats to display images in the **ImGearSilverlightPageView**, all others are optional. The developer gets to make the choice.

Listing 2: Page Constructor and ImageGear for Silverlight Initialization

```
public Page()
{
    InitializeComponent();

    // ImageGear for Silverlight Development licensing. This licensing
    // requires the sample to be run from a URL hosted on the same machine
    // ImageGear for Silverlight was installed on.
    string webServiceUrl = string.Format("http://{0}/{1}",
        App.Current.Host.Source.DnsSafeHost,
        "SilverlightWebService/SilverlightWebService.svc");

    ImGearLicense.SetService(webServiceUrl);
    ImGearLicense.LicenseRequest =
        new ImGearLicense.DelegateLicenseRequest(delegate(Exception error)
        {
            // Initialize format components
            // Required for ImGearSilverlightPageView
            ImGearFormatsPNGIFComponent.Initialize();
            ImGearFormatsWindowsComponent.Initialize();

            // Common formats
            ImGearFormatsCCITTComponent.Initialize();
            ImGearFormatsTIFFComponent.Initialize();
            ImGearFormatsJPEGComponent.Initialize();

            // ImageGear Initialized and Licensed; Enable the Open Button
            this.mOpenButton.IsEnabled = true;
        });

    ImGearLicense.SetSolutionName("Accusoft 5-44-16");
}
```

Once the toolkit is initialized, we can write handlers for each of our buttons, and get some images to display in the application. The **mOpenButton** handler (Listing 3) performs the bulk of the work. Most of the code is stock, and will be familiar to anyone who has utilized the **OpenFileDialog** type before. The **Stream** created is passed as an argument to **ImGearFileFormats.LoadDocument** to be loaded. If a codec corresponding to the file type is known to the toolkit (via initialization above), it will load; otherwise, an **ImGearException** is thrown.

With the document loaded and an instance of the **ImGearDocument** in hand, an **ImGearPageDisplay** instance is created to specify the viewable page to the **ImGearSilverlightPageView**. The easiest way to link the concepts is to think of the **ImGearDocument** as the in-memory representation of the file, and an **ImGearPageDisplay** as the logical representation of the page for display. In other words, the **ImGearPageDisplay** type contains the instructions for **ImGearSilverlightPageView** to display the page properly. Examples might include resolution, clipping rectangles, or transforms desired. Finally, to get the page to display, an **ImGearPageDisplay** is assigned to the **ImGearSilverlightPageView.Display** property, and a call to **ImGearSilverlightPageView.Update** made to force a redraw to occur.



Listing 3: Open Button Handler

```
private ImGearDocument mLoadedDocument = null;
private Int32 mCurrentPage = -1;
private ImGearPageDisplay mCurrentPageDisplay = null;

    .
    .
    .

private void mOpenButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    // Prompt the user to select an image file
    OpenFileDialog ofd = new OpenFileDialog();
    ofd.Filter =
        "Image files (*.tif;*.jpg;*.png;*.bmp)|*.tif;*.jpg;*.png;*.bmp|All files (*.*)|*.*";
    ofd.FilterIndex = 1;
    bool? result = ofd.ShowDialog();

    if (!result.GetValueOrDefault(false))
        return;

    try
    {
        using (Stream f = ofd.File.OpenRead())
        {
            // Open it with ImageGear, showing the first page by default
            this.mLoadedDocument = ImGearFileFormats.LoadDocument(f, 0, -1);
            this.mCurrentPage = 0;

            if (null == this.mCurrentPageDisplay)
            {
                this.mCurrentPageDisplay =
                    new ImGearPageDisplay(this.mLoadedDocument.Pages[this.mCurrentPage]);
            }
            else
            {
                this.mCurrentPageDisplay.Page =
                    this.mLoadedDocument.Pages[this.mCurrentPage];
            }

            mPageView.Display = this.mCurrentPageDisplay;
            mPageView.Update();

            // Update the previous\next buttons depending on page count
            this.UpdateButtonStates();
        }
    }
    catch (ImGearException)
    {
        MessageBox.Show("The file selected is not supported by this sample.",
            "MultiPageTiffViewer Control", System.Windows.MessageBoxButton.OK);
    }
}
```

The remainder of the code manipulates the page displayed via the previous and next buttons, modifying the page displayed by the **ImGearSilverlightPageView** type. The **Pages** collection within the **ImGearDocument** allows you to select the page loaded in the **ImGearPageDisplay**. Remember, when changing the page display, always call **ImGearSilverlightPageView.Update**, or a redraw will not occur.



Listing 4: Previous and Next Button Handlers, UpdateButtonStates Method

```
private void mPreviousButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    if (this.mCurrentPage > 0)
    {
        this.mCurrentPageDisplay.Page = this.mLoadedDocument.Pages[--this.mCurrentPage];
        this.mPageView.Update();
    }

    // Update the previous\next buttons depending on page count
    this.UpdateButtonStates();
}

private void mNextButton_Click(object sender, System.Windows.RoutedEventArgs e)
{
    if (this.mCurrentPage < this.mLoadedDocument.Pages.Count - 1)
    {
        this.mCurrentPageDisplay.Page = this.mLoadedDocument.Pages[++this.mCurrentPage];
        this.mPageView.Update();
    }

    // Update the previous\next buttons depending on page count
    this.UpdateButtonStates();
}

private void UpdateButtonStates()
{
    this.mPreviousButton.IsEnabled =
        (this.mCurrentPage > 0 && mLoadedDocument.Pages.Count > 1) ? true : false;

    this.mNextButton.IsEnabled =
        ((this.mCurrentPage < (this.mLoadedDocument.Pages.Count - 1) &&
        mLoadedDocument.Pages.Count > 1)) ? true : false;
}
```

And, with that, we have our multi-page image viewer for Silverlight! Figure 6 below shows the final product, with some “splash” applied by the web designer. With ImageGear for Silverlight, we’ve been able to extend the base platform to support loading of TIFF files, and we have a basic RIA suitable for document imaging display. Best of all, it was all done with current .NET technologies like XAML and C#, using tools in the ecosystem. Of course, a full featured viewer would do much more, and ImageGear for Silverlight will help you get there.

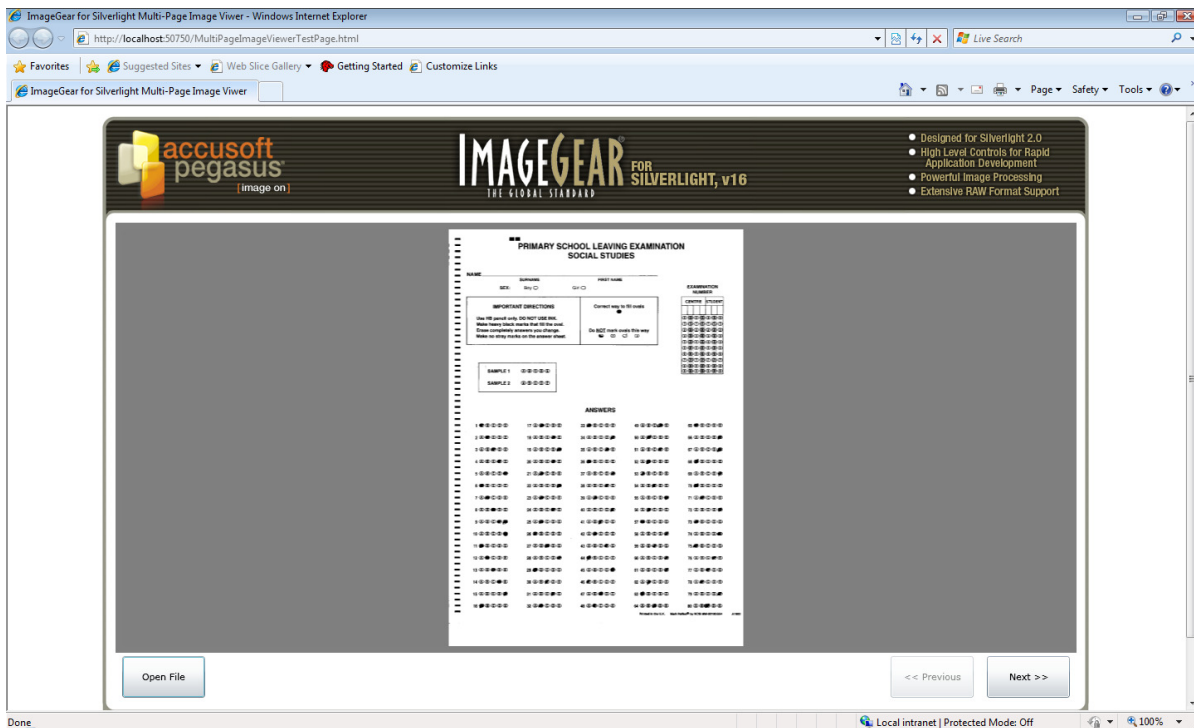


Figure 6: Multi-Page Image Viewer

You can find Accusoft product downloads and features at www.accusoft.com. Please contact us at sales@accusoft.com or support@accusoft.com for more information.

About Accusoft

Accusoft provides a full spectrum of document, content and imaging solutions. With its broad range of solutions, Accusoft is committed to deliver best-in-class, enterprise grade and fully-supported applications and a globally recognized suite of software development kits (SDKs). Accusoft products work reliably behind the scenes for capturing, processing, storing and viewing images, documents and more. Add barcode, compression, DICOM, image processing, OCR/ICR, forms processing, PDF, scanning, video, and image viewing to your applications. Products are delivered as applications and toolkits for multiple 32-bit/64-bit platforms and development environments, including iOS, Android, .NET, Silverlight, ASP.NET, ActiveX, Java, Linux, Solaris, Mac OSX, and IBM AIX. For more information, please visit www.accusoft.com.