Zero Footprint, Fast Viewing: Using an ASP.NET Imaging SDK to Build a Web-Based Image Viewer

As a market leader, Accusoft frequently communicates with developers and IT personnel about the issues they face. These are some of the statements we hear:

> "We have a lot of product manuals on our web site. We would like our customers to look at them in a viewer that matches the theme of our web site, rather than Adobe's Acrobat Reader."

> "I work for a service bureau, which scans documents for other companies. We are building a system that will let us host document management systems for our customers so we can enter the SaaS market."

> "We developed our own client-based content management system and now our users need to be able to quickly view PDF, TIFF, and JPEG documents over the web."

Does this sound like you? More and more the answer to that question is "yes." The tools and technologies that were once limited to a vertical market, the "content management system," are quickly becoming a part of daily life for most people.

This article discusses trends in software technologies and in the document management industry. Along the way, we will build an easy, yet feature-rich web-based application for viewing PDF documents. It will be based on our ASP.NET image-viewing technology, which is included in version 17 of our ImageGear for .NET product. You can download ImageGear for .NET from the Accusoft site.

Trends

This article discusses two major trends, giving details and examples: the trend toward browser-based, client-side development and the trend toward more powerful products that are simpler to use.

Many different efforts (HTML 5's canvas object, SVG, JavaScript performance advances, and hardware rendering) have converged in the past year to make it possible to build a sophisticated viewer that runs in a browser, with minimal support from a server. ImageGear for .NET still uses the server for things like reading PDF and JPEG 2000 images, but it does things like scaling, panning, and rotation in the browser so you end up with a very responsive application that places less burden on your server.

The software development industry is constantly pushing to do more with less development effort. The web is a significant step in that process, enabling you to easily build sophisticated applications that span the globe. Some of the nicest savings are that you do not need to create an installer (this is commonly referred to as a "zero footprint" application) and you can more easily target a wide variety of client platforms.

This is an exciting time in the application development world; change is in the air and the future looks brighter than ever. While change brings difficulties, in this case it also brings a wealth of

options for businesses and software developers. One of the best of those options is ASP.NET development. The focus of this article is the latest version of our ASP.NET image viewing technology, which is included in version 17 of our ImageGear for .NET product.

Description of the Application

One of the best ways to learn something is to "dive right in", so we're going to build a real, albeit simple, application. This application will let a user browse and view a collection of whitepapers, but it is not hard to imagine that the collection could be product manuals, engineering drawings, books, or medical insurance documents.

For virtually any system like this, the most important features are searching and viewing. Ultimately, all a user wants to do is to view the document(s) they are interested in. The search feature is critical, but is still only a means to an end: the user needs some way to tell the system which document they want to view.

While some kind of search feature is common to most systems like these, the details of that search feature can vary widely. For a system that provides product manuals, the user will usually know the model number or name of the product. For a system that provides medical insurance records, the user will usually know the id of a specific person and have a range of dates they are interested in. In our case, a user will usually have a topic that they are interested in. They could also have the name of an author or a product. For now, we will implement simple topic searching and leave the other options for a future version of the system.

While we could implement the system as a single web page with search and view modes, it will be easier to create separate search and view pages. The search page will display a list of keywords, along with thumbnails of the whitepapers that match the selected keyword. A user can click on any of the keywords in the list and the page will instantly filter the library of whitepapers to match. We will define a special keyword, "Recent," and set that as the initial keyword. A screen capture is below, showing the initial state with the "Recent" keyword selected.
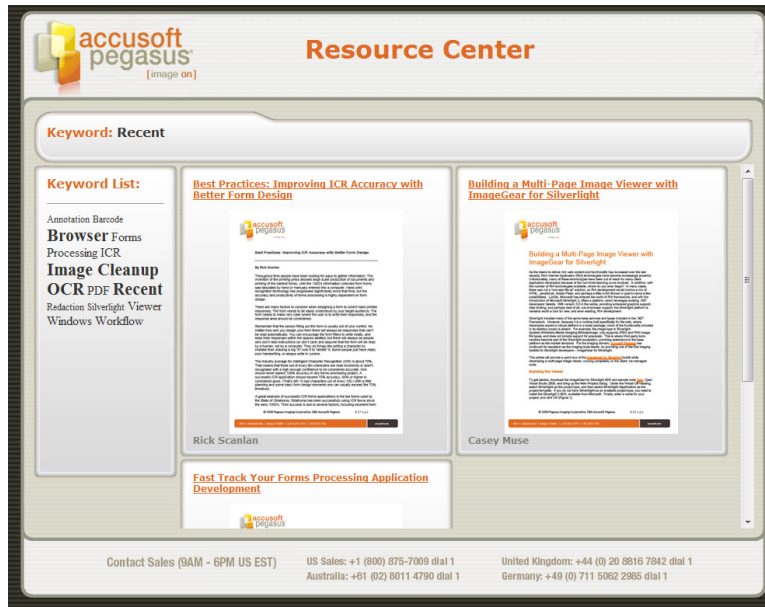
Figure 1: Search Page

For the view page, we will go with a fairly clean user interface, with a single page of the whitepaper visible, along with a set of buttons along the bottom edge. For a robust application, you would want some optional modes, like one where two pages are visible at a time (often called a "2-up" view), or perhaps a list of thumbnails for each page.
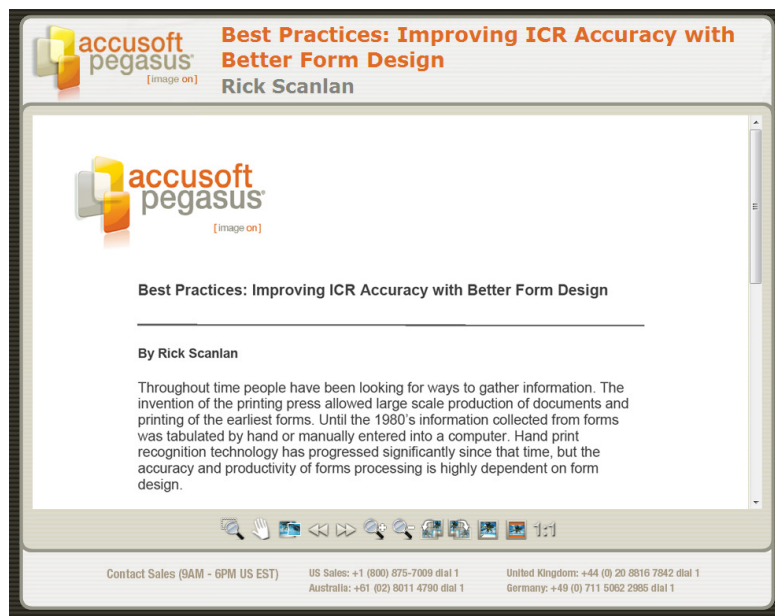

Figure 2: View Page

Building the Application

Before we get to work, you should download and install ImageGear for .NET, version 17. You can download an evaluation version on the Accusoft web site. After you install it, you should go through the tutorial in the "Getting Started" section of volume 1 of the ImageGear help file (the installer will place a link to it on your Start Menu.) That tutorial will show you how to license and deploy your application. This article will not duplicate that information.

We will begin the coding with the view page. I have made the page fixed-size so we can focus on the viewer instead of CSS, but you should consider using a fluid layout.

We will name the page View.aspx and the content is below.

Listing 1: View.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="View.aspx.cs"
    Inherits="View" %>
<%@ Register Assembly="ImageGear17.Web" Namespace="ImageGear.Web.UI"
    TagPrefix="imgear" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Resource Center Document Viewer</title>

    <script src="js/jquery-1.3.2.js" type="text/javascript"></script>
    <script src="js/View.js" type="text/javascript"></script>
    <link href="css/Common.css" rel="stylesheet" type="text/css" />
    <link href="css/View.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div class="Application">
        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <div id="PageHeader">
            <img class="SectionLeft" src="images/HeaderLeft.gif" />
            <div class="SectionCenter">
                <img src="images/HeaderLogo.jpg" />
                <span id="PageHeaderTitle1" runat="server"></span>
                <span id="PageHeaderTitle2" runat="server">
                </span>
            </div>
            <img class="SectionRight" src="images/HeaderRight.gif" />
        </div>
        <div id="PageBody">
            <img class="SectionLeft" src="images/BodyLeft.gif" />
            <div class="SectionCenter">
                <imgear:ImageView ID="Viewer" runat="server"
                    CssClass="Viewer" />
                <div class="Toolbar">
                    <img src="images/ButtonSelectAndZoom.png"
```

```
                    id="ActionSelectAndZoom" />
                <img src="images/ButtonHandPan.png"
                    id="ActionHandPan" />
                <img src="images/ButtonMagnifier.png"
                    id="ActionMagnifier" />
                <img src="images/ButtonPreviousPage.png"
                    id="ActionPreviousPage" />
                <img src="images/ButtonNextPage.png"
                    id="ActionNextPage" />
                <img src="images/ButtonZoomIn.png"
                    id="ActionZoomIn" />
                <img src="images/ButtonZoomOut.png"
                    id="ActionZoomOut" />
                <img src="images/ButtonRotateLeft.png"
                    id="ActionRotateLeft" />
                <img src="images/ButtonRotateRight.png"
                    id="ActionRotateRight" />
                <img src="images/ButtonFullImage.png"
                    id="ActionFullImage" />
                <img src="images/ButtonFullWidth.png"
                    id="ActionFullWidth" />
                <img src="images/ButtonActualSize.png"
                    id="ActionActualSize" />
            </div>
        </div>
        <img class="SectionRight" src="images/BodyRight.gif" />
    </div>
    <div id="PageFooter">
        <img class="SectionLeft" src="images/FooterLeft.gif" />
        <div id="FooterCenter" class="SectionCenter">
            <img src="images/FooterText.gif" />
        </div>
        <img class="SectionRight" src="images/FooterRight.gif" />
    </div>
    </form>
</body>
</html>
```

The most important part of this page is the ImageView control, which will construct a client-side control that lets you view images. The single most important task for the view page is to open the correct document. We are going to get that process started by opening the first page of the document in the code-behind for the page.

Listing 2: View.aspx.cs

```
using System;
using System.Linq;
using System.Web.Configuration;

public partial class View : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
```

```csharp
        string cs = WebConfigurationManager
            .ConnectionStrings["ImageDatabaseConnectionString"]
            .ConnectionString;
    using (ImageDatabase db = new ImageDatabase(cs))
    {
        // Pull the database ID of the requested document out of
        // the query string.
        Guid Identifer = new
            Guid(Request.QueryString["Identifier"]);

        // Get the full information for the specified document.
        var doc =
            (from document in db.Documents
            where document.DocumentID == Identifer
            select document).First();

        // Place the document title and author in the page header.
        PageHeaderTitle1.InnerText = doc.Title;
        PageHeaderTitle2.InnerText = doc.Author;

        // Tell the viewer which document to load.
        Viewer.ImageIdentifier = doc.DocumentID.ToString();
    }
    }
}
```

We will use the query string to indicate which document to view. Obviously, there are a variety of other options for passing that information to the page. The last three lines of code are the most important, setting the page title and the ImageIdentifier property. Since this application keeps the documents in a database, you need to set the image identifier to the key field of the table that holds the documents.

While this may seem too simple, it's really not. All that ImageGear for .NET needs is the database key and a small amount of configuration in web.config (which we will get to toward the end of this article.) ImageGear for .NET includes an ASP.NET handler that will query the database to get your images. It also automatically figures out what file type the document is (PDF, TIFF, JPEG, etc.) and handles is appropriately.

We need a little CSS to lay out the page. Part of the CSS is common to both pages, so we will put that into a separate file called Common.css. This CSS is needed to lay out the headers, body, and footer of the page. It is not directly related to the topic of this article, so I have omitted it to save space.

The rest of the CSS goes into a file called View.css. It is simply indicating the position, size and font to use for the various elements of the view page.

Listing 3: View.css

```css
.Viewer {
    position:absolute;
    width:928px;
    height:500px;
```

```
}
.Toolbar {
    position:absolute;
    top:510px;
    left:240px;
}
#PageHeaderTitle1 {
    position:absolute;
    left:240px;
    top:8px;
    font:normal normal bolder 26px Verdana;
}
#PageHeaderTitle2 {
    position:absolute;
    left:240px;
    top:76px;
    font:normal normal bolder 24px Verdana;
    color:#83847B;
}
```

Finally, the meat of this page is the JavaScript that will run the viewer. We are using jQuery to make it easier. Speaking of jQuery, it is one of those trends I mentioned earlier. Nowadays, most JavaScript development relies on one or more JavaScript toolkits. jQuery is one of the most popular, especially for ASP.NET developers since Microsoft formally supports it. jQuery does a great job of hiding the differences between browsers so you can focus on more important issues. It also has some powerful, yet easy to use functions for animating and adjusting your HTML.

I am a big fan of jQuery because it makes the browser so much easier to manage. Whether you use jQuery or some other JavaScript library, there is a clear trend toward building web pages that rely on other libraries, such as jQuery, to make life easier.

jQuery is also an example of another trend, more powerful API's that are easier to use. As you look at the JavaScript below, it is obvious that the ImageView control embraces that trend. As we designed the API, we tried very hard to look at it from your perspective and match the API to your needs. Like jQuery, we are doing a lot of difficult stuff under the hood, but we tried our best to hide that so you can focus on the rest of your application.

Listing 4: View.js
```javascript
// This variable will be an alias for the ImageGear namespace.
// Using an alias makes the rest of the JavaScript shorter.
var ig;

// Handle the toolbar click events.
function toolbarClickHandler() {
    var cs;
    // Use the ASP.NET 3.5 function, $find(), to get a reference to the
    // ImageView client-side control.
    var iv = $find('Viewer');

    switch (this.id) {
```

```
case 'ActionSelectAndZoom':
    // Set the viewer's left-mouse-button tool to zoom in on a
    // selected rectangle. The user will be able to click and
    // drag a rectangle. When the user releases the mouse
    // button, the viewer will zoom in on the selected area.
    iv.set_mouseTool(ig.MouseTool.RectangleZoom);
    break;

case 'ActionHandPan':
    // Set the viewer's left-mouse-button tool to pan the
    // image. The user will be able to click and drag the
    // image.
    iv.set_mouseTool(ig.MouseTool.HandPan);
    break;

case 'ActionMagnifier':
    // Set the viewer's left-mouse-button tool to show the
    // magnifying glass. When the user clicks the mouse button,
    // the viewer will display a magnifying glass. The
    // magnifying glass will follow the cursor until the user
    // releases the mouse button.
    iv.set_mouseTool(ig.MouseTool.Magnifier);
    break;

case 'ActionPreviousPage':
    if (iv.get_imageIsOpen()) {
        // Get the current state of the viewer.
        // The current state includes the page index of the
        // most-recently-opened image.
        cs = iv.get_currentState();

        if (cs.imagePageIndex > 0) {
            // Open the previous page, using the full width of
            // the viewer.
            iv.openImage({
                imageIdentifier: cs.imageIdentifier,
                imagePageIndex: cs.imagePageIndex - 1,
                viewFitType: ig.FitType.FullWidth
            });
        }
    }
    break;

case 'ActionNextPage':
    if (iv.get_imageIsOpen()) {
        // Get the current state of the viewer.
        // The current state includes the page index of the
        // most-recently-opened image.
        cs = iv.get_currentState();

        if (cs.imagePageIndex < iv.get_imagePageCount() - 1) {
            // Open the next page, using the full width of the
```

```
                    // viewer.
                    iv.openImage({
                        imageIdentifier: cs.imageIdentifier,
                        imagePageIndex: cs.imagePageIndex + 1,
                        viewFitType: ig.FitType.FullWidth
                    });
                }
            }
            break;

        case 'ActionZoomIn':
            if (iv.get_imageIsOpen()) {
                // Zoom in 50%.
                // The portion of the image in the center of the viewer
                // will remain in the center of the viewer. The user
                // can also zoom in using control-mouse-wheel and the
                // portion of the image under the cursor will remain in
                // the same position.
                iv.zoomIn(1.5);
            }
            break;

        case 'ActionZoomOut':
            if (iv.get_imageIsOpen()) {
                // Zoom out 50%.
                // The portion of the image in the center of the viewer
                // will remain in the center of the viewer. The user
                // can also zoom out using control-mouse-wheel and the
                // portion of the image under the cursor will remain in
                // the same position.
                iv.zoomOut(1.5);
            }
            break;

        case 'ActionRotateLeft':
            if (iv.get_imageIsOpen()) {
                // Rotate counter-clockwise 90 degrees.
                // The portion of the image in the center of the viewer
                // will remain in the center of the viewer.
                iv.rotate(-90);
            }
            break;

        case 'ActionRotateRight':
            if (iv.get_imageIsOpen()) {
                // Rotate clockwise 90 degrees.
                // The portion of the image in the center of the viewer
                // will remain in the center of the viewer.
                iv.rotate(90);
            }
            break;
```

```javascript
        case 'ActionFullImage':
            if (iv.get_imageIsOpen()) {
                // Display the full image in the viewer's area.
                iv.fitImage(ig.FitType.FullImage);
            }
            break;

        case 'ActionFullWidth':
            if (iv.get_imageIsOpen()) {
                // Adjust the scale to use the full width of the
                // viewer's area. The top of the image will be scrolled
                // into view, but the bottom may not be visible. If the
                // image is currently rotated, it will remain rotated.
                iv.fitImage(ig.FitType.FullWidth);
            }
            break;

        case 'ActionActualSize':
            if (iv.get_imageIsOpen()) {
                // Display the image actual size. The top-left corner
                // of the image will be scrolled into view, but the
                // right and bottom may not be visible. If the image is
                // currently rotated, it will remain rotated.
                iv.fitImage(ig.FitType.ActualSize);
            }
            break;
    }
}

// This jQuery function will run when the page is ready.
// It is similar to the ASP.NET 3.5 pageLoad() function.
$(function() {
    // Create the ImageGear namespace alias.
    ig = ImageGear.Web.UI;

    // Using jQuery, assign an event handler to each toolbar button.
    // Handling double-click makes the button behave properly when a
    // user clicks quickly in IE.
    $('.Toolbar > img')
        .click(toolbarClickHandler)
        .dblclick(toolbarClickHandler);
});
```

There is really not much to the JavaScript. All you need to do is connect your user interface to the viewer and you are ready to go.

The search page will be a little more interesting. We are going to generate most of it on the client without doing page refreshes or using the ASP.NET UpdatePanel. One of the most exciting and promising trends in web development is the trend toward building full-blown applications that run in the browser, with a little assistance from the server. This is sometimes called a "Rich Internet Application," or RIA. Even without a server, modern web browsers are

fast enough and powerful enough to be considered a development environment just like Windows, Linux, or Mac. The search page will be exactly that kind of application-in-a-page.

ASP.NET 3.5 introduced a robust client-side framework for building controls and ImageGear for .NET fully embraces that client-centric approach. This really makes it easy to build client applications without the full or partial page refreshes that more traditional web applications require. ASP.NET MVC goes one step further and eliminates server-side web form controls. Since ImageGear for .NET includes a stand-alone client-side control, you can use it with ASP.NET MVC almost as easily as you can with ASP.NET WebForms.

Once again, let's start with the page. Call it Search.aspx and put this content into it.

Listing 5: Search.aspx

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Search.aspx.cs"
    Inherits="Search" %>
<%@ Register Assembly="ImageGear17.Web" Namespace="ImageGear.Web.UI"
    TagPrefix="imgear" %>

<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Resource Center</title>
    <script src="js/jquery-1.3.2.js" type="text/javascript"></script>
    <script src="js/Search.js" type="text/javascript"></script>
    <link href="css/Common.css" rel="stylesheet" type="text/css" />
    <link href="css/Search.css" rel="stylesheet" type="text/css" />
</head>
<body>
    <form id="form1" runat="server">
    <div>
        <asp:ScriptManager ID="ScriptManager1" runat="server"
            EnablePageMethods="true" />
        <div id="PageHeader">
            <img class="SectionLeft" src="images/HeaderLeft.gif" />
            <div class="SectionCenter">
                <img src="images/HeaderLogo.jpg" />
                <span id="PageHeaderTitle1">Resource Center</span>
            </div>
            <img class="SectionRight" src="images/HeaderRight.gif" />
        </div>
        <div id="PageBody">
            <img class="SectionLeft" src="images/BodyLeft.gif" />
            <div class="SectionCenter">
                <div id="Header">
                    <div>Keyword:
                        <span id="SelectedKeyword"></span>
                    </div>
                </div>
                <div id="Keywords">
                    <p id="KeywordsTitle">Keyword List:</p>
```

```
            <hr/>
            <asp:ListView runat="server" ID="KeywordListView">
                <LayoutTemplate>
                    <div id="KeywordList">
                        <p runat="server"
                            id="itemPlaceholder"/>
                    </div>
                </LayoutTemplate>
                <ItemTemplate>
                    <span class='<%# Eval("FontClass") %>'>
                        <%# Eval("Keyword") %></span>
                </ItemTemplate>
            </asp:ListView>
        </div>
        <div id="Thumbnails">
            <div></div>
        </div>
    </div>
    <div id="ThumbnailTemplate" style="display:none">
        <imgear:ThumbnailView ID="ThumbnailViewTemplate"
            runat="server" CssClass="ThumbnailView"/>
        <div class="Thumbnail">
            <div>
                <a class="ThumbnailTitle"></a>
                <div class="ThumbnailImage"></div>
                <div class="ThumbnailAuthor"></div>
            </div>
        </div>
    </div>
    <img class="SectionRight" src="images/BodyRight.gif" />
</div>
<div id="PageFooter">
    <img class="SectionLeft" src="images/FooterLeft.gif" />
    <div id="FooterCenter" class="SectionCenter">
        <img src="images/FooterText.gif" />
    </div>
    <img class="SectionRight" src="images/FooterRight.gif" />
</div>
</div>
</form>
</body>
</html>
```

There are several interesting things in this page. The first is the ListView control that will represent our keywords. It will be generated on the server before the page is sent to the browser, but we will get to that later. The next most interesting thing is that the center part of the page (the Thumbnails div) is empty; we will be updating that part of the page solely on the client. Each document that matches the selected keyword will be displayed in a block and the format of that block is defined by the template defined in the Thumbnail div. The final interesting feature of this page is the ThumbnailView control. We will not be using this ThumbnailView control in the client-side page; it is there simply to make sure the JavaScript for the

ThumbnailView gets downloaded to the browser. It would have been almost as easy to capture that JavaScript using FireBug or Fiddler and store it in a separate file. All the ThumbnailView controls we will use will be constructed in JavaScript code on the browser.

Next, we need some CSS to lay out the page. Most of this is just setting the sizes, colors, and positions of the various elements of the page. Of some interest are the four keyword styles. Those styles will be what distinguish common keywords from uncommon ones.

Listing 6: Search.css

```css
.Viewer {
    display:none;
}
#PageHeaderTitle1 {
    position:absolute;
    left:350px;
    top:30px;
    font:normal normal bolder 32px Verdana;
}
#Header {
    width:928px;
    height:70px;
    background-image:url(../images/SearchHeader.gif);
}
#Header > div {
    position:absolute;
    left:18px;
    top:20px;
    font:normal normal bold 16px Verdana;
}
#Header #SelectedKeyword {
    color:#373534;
}
#Keywords {
    position:absolute;
    left:0px;
    top:70px;
    padding: 0px 18px;
    float: left;
    width: 152px;
    height: 405px;
    overflow: auto;
    background-image: url(../images/SearchKeywordList.gif);
}
#KeywordsTitle {
    font:normal normal bold 16px Verdana;
}
#KeywordList {
    color:#373534;
}
#Thumbnails {
    position:absolute;
    left:188px;
```

```css
    top:70px;
    width:740px;
    height:469px;
    overflow: auto;
}
.Thumbnail {
    position:relative;
    float:left;
    width:354px;
    height:378px;
    background-image:url(../images/SearchThumbnail.gif);
    font:normal normal bold 12px Verdana;
}
.Thumbnail > div {
    padding:18px 18px;
}
.ThumbnailTitle
{
    display:block;
    height:40px;
}
.ThumbnailImage {
    height:290px;
}
.ThumbnailAuthor {
    color: #83847B;
    height:30px;
}
.HugeKeyword {
    font-size:28px;
    font-weight:800;
}
.LargeKeyword {
    font-size:22px;
    font-weight:600;
}
.MediumKeyword {
    font-size:16px;
    font-weight:400;
}
.SmallKeyword {
    font-size:12px;
    font-weight:200;
}
.Keyword {
    cursor:pointer;
}
.Keyword:hover {
    text-decoration: underline;
}
```

In order to understand the page, you need to see the schema of our database. It is really quite simple; we have two main tables: one for whitepapers and one for keywords. There is a third table that handles the many-to-many relationship between keywords and whitepapers. Here is an E-R diagram:
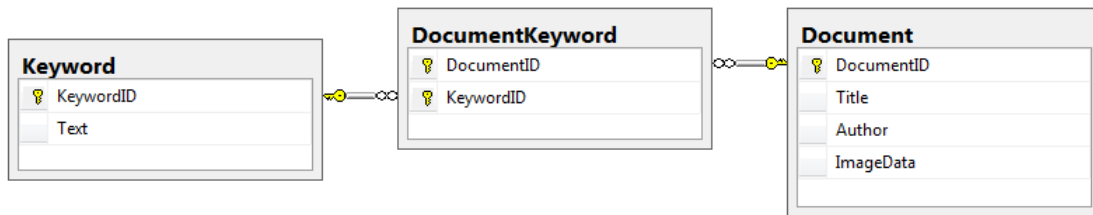
Figure 3: Entity-Relationship Diagram

Before we get to the fun stuff, let's take care of that keyword box. It would not be difficult to generate it in the browser, but it will not change during the life of the page, so we might as well create it on the server. Here's the C# code to do that.

Listing 7: Search.aspx.cs (part 1)

```csharp
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Web.Configuration;

public partial class Search : System.Web.UI.Page
{
    // This function converts a count of documents into a CSS class.
    private string FontClass(int count, int small, int medium,
        int large)
    {
        if (count > large)
            return "Keyword HugeKeyword";
        else if (count > medium)
            return "Keyword LargeKeyword";
        else if (count > small)
            return "Keyword MediumKeyword";

        return "Keyword SmallKeyword";
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        string cs = WebConfigurationManager
            .ConnectionStrings["ImageDatabaseConnectionString"]
            .ConnectionString;
        using (ImageDatabase db = new ImageDatabase(cs))
        {
```

```csharp
        // Get the keywords and the number of associated documents.
        var keywords =
            from relation in db.DocumentKeywords
            group relation by relation.Keyword into grouping
            select new
            {
                Keyword = grouping.Key,
                Count = grouping.Count()
            };

        // Sort the keywords in descending order by the count.
        var sortedKeywords =
            from keyword in keywords
            orderby keyword.Count descending
            select keyword;

        // Pick the thresholds between the four different sized
        // fonts. One fourth of all the keywords will be huge, one
        // fourth large, etc.
        var arrayOfKeywords = sortedKeywords.ToArray();
        var length = arrayOfKeywords.Length;
        int large = arrayOfKeywords[length / 4].Count;
        int medium = arrayOfKeywords[length * 2 / 4].Count;
        int small = arrayOfKeywords[length * 3 / 4].Count;

        // Sort the keywords in alphabetical order and add the CSS
        // class for each font size.
        var keywordsAndFonts =
            from keyword in sortedKeywords
            orderby keyword.Keyword.Text
            select new
            {
                Keyword = keyword.Keyword.Text,
                FontClass = FontClass(keyword.Count, small, medium,
                    large)
            };

        // Data bind the list to the ListView control.
        KeywordListView.DataSource = keywordsAndFonts;
        KeywordListView.DataBind();
    }
  }
}
```

The code is using LINQ to SQL to get a list of keywords and the number of whitepapers that go with each keyword, sorted by the count. Next we are using a little procedural code to determine how many matching documents are required for each font size, with larger fonts being assigned to the keywords that have higher counts of associated documents. The actual font size is set with CSS. Finally, we are sorting the keywords in alphabetical order and data-binding them to the keyword ListView. The ListView control will take care of the rest of the work.

Now we need some JavaScript. While our page is more sophisticated than a classic web page, it really just needs two things: a function to update the page to match a selected keyword and a way to navigate to the viewer page.

The function to update the page is the most complex. It needs to build up the HTML to display the thumbnail blocks by making copies of the thumbnail block template and it needs to create ThumbnailView controls to display the first page of each image. But the most uniquely web part of the function is that it needs to ask the web server for a list of documents that match the keyword. We will implement that in the form of an ASP.NET page method so that ASP.NET will automatically create a JavaScript proxy to let us call it. We could make that Ajax call using jQuery, but using ASP.NET is easier.

Listing 8: Search.js (part 1)

```javascript
// This variable will be an alias for the ImageGear namespace.
// Using an alias makes the rest of the JavaScript shorter.
var ig;

// This variable holds the most recently selected keyword. It is
// possible for a user to click several keywords then have the page
// methods finish out of order. This variable solves that problem by
// letting the completion function ignore any returned results other
// than the last one.
var selectedKeyword;

// This variable holds the absolute URL back to the server so the
// ThumbnailView can request images.
var imageHandlerUrl;

// This variable holds a number that increments each time a control is
// created. The value of the number is appended to the ID of the
// control so each has a unique ID.
var controlCounter = 0;

function selectKeyword(keyword)
{
    // Save the new keyword so any other pending select operations can
    // cancel themselves.
    selectedKeyword = keyword;

    var prepareThumbnails = function(result)
    {
        if (keyword !== selectedKeyword)
        {
            // This is not the last selected keyword, so just ignore
            // the result. The user must have clicked on more than one
            // keyword before the server responded.
            return;
        }

        // Use jQuery to create a new div.
        // All the new thumbnails will go into that div.
```

```javascript
        var newThumbnails = $('<div></div>');
        var thumbnail;

        $(result).each(function() {
            thumbnail = $('#ThumbnailTemplate > .Thumbnail').clone();
            thumbnail.find('.ThumbnailTitle')
                .text(this.title)
                .attr('href', 'View.aspx?Identifier=' +
                this.documentID);
            var controlDiv = thumbnail.find('.ThumbnailImage').get(0);
            controlDiv.id = 'ThumbnailView' +
                (controlCounter++).toString();
            controlDiv.imageIdentifer = this.documentID;
            thumbnail.find('.ThumbnailAuthor').text(this.author);
            newThumbnails.append(thumbnail);
        });

        // Dispose of all the existing ThumbnailView controls (if any.)
        $('#Thumbnails .ThumbnailImage').each(function() {
            if (this.control) {
                this.control.dispose();
            }
        });

        // Replace the existing thumbnails with a new list
        $('#Thumbnails > div').replaceWith(newThumbnails);

        // Create and open the ThumbnailView controls.
        $('#Thumbnails .ThumbnailImage').each(function() {
            if (this.imageIdentifer) {
                $create(ig.ThumbnailView,
                    { imageHandlerUrl: imageHandlerUrl },
                    null,
                    null,
                    this);
                this.control.openImage({
                    imageIdentifier: this.imageIdentifer,
                    imagePageIndex: 0
                });
            }
        });

        // Place the keyword into the header.
        $('#SelectedKeyword').text(keyword);
    }

    // Call the page method to get a list of documents that match the
    // keyword. The page method is running on the server, so ASP.NET
    // will use Ajax to call it. When the page method finishes, ASP.NET
    // will call prepareThumbnails().
    PageMethods.GetDocumentsForKeyword(keyword, prepareThumbnails);
}
```

It sounds like a lot of work, but the code is not that bad. The function starts by saving the selected keyword, then it calls our ASP.NET page method, GetDocumentsForKeyword(). We will get to the code for that method soon. Saving the keyword is important to avoid a subtle problem that is common in Ajax applications. Calls to the server take some time, and run asynchronously, so a user may click on several keywords before the first request comes back to the server. One option would be to disable the keyword selection while a request is outstanding, but it is better for subsequent clicks to override the previous operation. Saving the most recently selected keyword makes it easy to ignore those earlier operations when they finish.

Once the page method completes, the prepareThumbnails() function uses jQuery to duplicate the thumbnail template once for each matching document. Next, we dispose of any existing ThumbnailView controls and swap the existing page content with the new thumbnails. Finally, we use the ASP.NET client framework to create our ThumbnailView controls and open them to the first page of the document.

Next, we need to create the ASP.NET page method that the client uses to determine which pages match a given keyword. This amounts to a simple database query. Again, we are going to use LINQ to SQL.

Listing 9: Search.aspx.cs (part 2)

```csharp
[System.Web.Services.WebMethod]
public static object GetDocumentsForKeyword(string keyword)
{
    string cs = WebConfigurationManager
        .ConnectionStrings["ImageDatabaseConnectionString"]
        .ConnectionString;
    using (ImageDatabase db = new ImageDatabase(cs))
    {
        // Get the documents that match the given keyword.
        var documents =
            from row in db.DocumentKeywords
            where row.Keyword.Text == keyword
            orderby row.Document.Title
            select new
            {
                documentID = row.Document.DocumentID,
                title = row.Document.Title,
                author = row.Document.Author
            };

        // Return the array of matching documents.
        return documents.ToArray();
    }
}
```

The last code we need is the code that will run the search page. We have the function that initializes the page and the event handler that the browser calls when a user selects a keyword.

Listing 10: Search.js (part 2)

```javascript
// Handle the keyword click events.
function keywordClickHandler() {
    selectKeyword($(this).text());
}

// ASP.NET 3.5 will call this function when the page is loaded and
// ready to be scripted.
function pageLoad() {
    // Create the ImageGear namespace alias.
    ig = ImageGear.Web.UI;

    // Get the absolute URL back to the image handle, running on the
    // server. This has to be an absolute (due to a problem in FireFox
    // 3.0) URL back to the image handler running on the server.
    imageHandlerUrl = $find('ThumbnailViewTemplate')
        .get_imageHandlerUrl();

    // Set the default keyword to "Recent".
    selectKeyword('Recent');

    // Using jQuery, assign an event handler to each keyword.
    $('.Keyword').click(keywordClickHandler);
}
```

We are done with all the code; the only thing remaining is to configure ImageGear for .NET's web.config settings to read images from the database. You simply need to reconfigure the default SqlImageDataProvider to read from the database. Specifically, that involves setting the sqlCommand to "SELECT ImageData FROM dbo.Document WHERE [DocumentID] = @Image_key".

Our SQL configuration allows you to use nearly any schema for your database. We have a lot of customers with pre-existing databases, so asking someone to change their database to match the product can be a show-stopper. There is also a way to handle more exotic storage mechanisms, but that is beyond the scope of this whitepaper.

Summary

Well, there you have it: a simple web application for viewing PDF documents. Obviously, the application can match whatever theme you select for your web site. The best part is that very little of the code is related to the primary task: document viewing. ImageGear has a ton of code for document viewing and you just need to connect it to your application.

Download it today at Accusoft and try it for yourself. It is powerful, easy to use, and its architecture lets it embrace the latest development trends.

About The Author
Butch Taylor, Sr. Software Engineer, Accusoft

Butch Taylor has a passion for solving complex engineering challenges. He joined Accusoft (Pegasus Imaging) with the acquisition of TMSSequoia in December 2004. As an important member of the team since 1987, Butch has contributed many lines of code and applied expertise to several high performance document imaging product lines from Accusoft, such as ImageGear, Prizm Viewer, FormFix, PICTools, and ScanFix. His primary goal is applying new technologies to expand opportunities in imaging, and his talent is exposed in his code optimization and image analysis work related to document image cleanup and forms processing. He finds it personally rewarding to solve customer problems, and insists on staying customer-focused. Butch has contributed to patents both earned and pending, and received both a Masters and Bachelor of Science in Computer Science from Oklahoma State University.

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.