Creating a ASP.NET Contact Form

When developing a website for business, personal or organizational use, having a reliable, secure and intuitive way of ascertaining information from your visitors in an effort to better meet their needs, creating and developing a contact form is essential. Creating a form allows website owners to ask for specific information from the visitor in an effort to respond back to them in a way that seems meaningful and useful. However, if precautions are not taken, creating these forms can lead to abuse, misuse and incorrect information getting back to the business, person or organization, and in most cases, leading to a mess of unsolicited electronic mail that wreaks havoc on email systems.

In this article, we will look at creating a ASP.NET contact form that will allow us to capture specific information that we need, validate form fields using client and server side languages to ensure that fields are not empty and then send the results to a specified recipient (s) that will allow them to respond to the visitor in depth when time and information permits. In order to do this, we need to download and install the .NET framework, install Internet Information Services (IIS), understand client-server architecture and its relation to web programming. If you would like to learn how to create such a form, as well as learning about these concepts in-depth please follow along with the article below.

Requirements

In order for us to work effectively, we'll create our form through Internet Information Services (IIS) on our local computer and then transfer the files to our remote host. There are a variety of reasons for taking this approach including: (1) some hosts have low remote connectivity timeouts which makes developing these types of forms difficult to accomplish remotely, (2) you have more flexibility debugging your code locally and (3) most hosts don't allow direct access to their server from within an editor. As a result, we'll need the following software installed on our computer:

- Windows XP Professional
- Internet Information Services (IIS)
- Microsoft.NET Framework (version 2.0 or 3.5 will work)
- Preferably Visual Web Developer express (free)

Pre Download Information

For the purposes of this article, it's assumed you have Windows XP Professional installed. Continuing, let's first download and install the .NET framework from Microsoft. If you're wondering why we're skipping Internet Information Services (IIS) at the moment, it's because often times if you install IIS before installing the .NET framework, IIS will not install and configure properly some of its services correctly, including the association it has to the .NET framework, among others.

Download and Install .NET Framework

In order to download and install the .NET framework, open your preferred web browser and follow the web address below:

http://www.asp.net/downloads/essential/

From this page, you'll see two download areas, one for the .NET framework and one for Visual Web Developer Express, choose the .NET framework. Once you click the link, you'll be prompted to save the executable file to a location of your choice, once downloaded, double click the file and follow the steps.

Download and Install Visual Web Developer Express

In order to download Visual Web Developer Express, return to the web address provided above, click the download link, and once again, you'll be prompted to save the executable file to a location of your choice, once downloaded, double click the file and follow the steps.

Installing Internet Information Services (IIS)

You may be wondering at this point, if we need to bother with installing Internet Information Services (IIS) since we already installed the .NET framework and Visual Web Developer. Even though we have the framework installed which gives us access to the classes, web controls and resources needed to create our form, when working with web applications we still need a web server that is capable of accepting our request from the client's browser, parsing server side web controls, programmatic code and then sending the result back in html so that a web browser can understand and render the web page appropriately. More on the concept client-server architecture will be covered later in the article.

In order to install Internet Information Services (IIS), follow these steps:

- From your desktop, click Start : Settings : Control Panel
- In your Control Panel, double click *Add or Remove Programs*
- In this window, on the left side, left click *Add/Remove Windows Components*
- In the Windows Components Wizard, scroll down until you find *Internet Information Services* and check the box
- Left click *Next*
- At some point during the install process you'll be prompted for your Windows XP CD which is where IIS resides

Once this is complete, close all windows, we'll need to test and verify the ASP client was registered with IIS.

Testing Local host (IIS)

Before continuing, let's ensure IIS is installed and configured properly to handle ASP.NET. Open your preferred web browser and type the following address:

http://localhost/

If you get an error screen, don't panic. Sometimes during the installation and configuration of the .NET framework & IIS, the ASP client isn't registered with IIS. It's nothing to fret about and can be easily fixed by opening a command line and typing the following:

C:\WINDOWS\Microsoft.NET\Framework\[version]\aspnet_regiis -i

Where version is the version of the .NET framework installed on your computer. Once this is complete, refresh your browser and a welcome screen should show.

Create a Virtual Directory/Application

In the ASP.NET environment, when creating web forms, you'll want to create them within a virtual directory or application in IIS. Associating web forms to a virtual directory or application within IIS allows your application to operate more efficiently and allows your application access to features that makes working with ASP.NET web forms far easier than most server side environments. Such features include:

- Access to a Web.config file that allows your application to contain information including:
    - Connection string information to a database, default compilation settings such as programming languages (C# or VB.NET), registration of in-house written components and authentication settings.
- Creation of two Web form files:
    - WebForm.aspx
    - WebForm.aspx.cs

More on the intricacies of these files will be discussed later in the article.

In order to create a virtual directory or application in IIS, you need a physical folder setup on your hard drive. Once completed, follow the steps below to open IIS:

- From the desktop, click Start : Settings : Control Panel
- In the Control Panel window, double click *Administrative Tools*
- In the Administrative Tools window, double click *Internet Information Services*

- In the Internet Information Services window, expand the plus sign next to your computer name, which should show two folders, *Web Sites* and *Default SMTP Server*
- Expand the plus sign next to Web Sites, which should show *Default Web Site*.
- To create the virtual directory/application, right click on Default Web Site and select New : Virtual Directory
- In the Virtual Directory Wizard window, click *Next*
- For the alias name, type *dev*, which is what you'll use to identify your virtual directory/application, and then, click *Next*
- For the directory, click *Browse* and select your physical root folder where you'll be creating the web form
- For the access permissions, leave as is, and click *Next*, which should show a finished window, click *Finish*
- In the IIS window, you should see your newly created virtual directory/application as well as the files it contains. The directory should be empty.

Close the IIS window and open Visual Web Developer Express, we're ready to begin creating our contact form.

Open Local Web Site from Visual Web Developer Express

When you first open Visual Web Developer Express, a start up screen will show with a default language setting. For the purposes of this article, we'll be using C# (pronounced C-sharp). To open our web application, follow these steps:

- From the main menu, select File : Open Web Site
- In the Open Web Site window, on the left side, select *Local IIS*
- You should see the virtual directory/application you just created, left click to select it and left click *Open*

Creating the Web Form

With the local web site open, you should see your virtual directory/application in the upper right side of the screen. In order to create our contact form, complete the following steps:

- Right click your application and select *Add New Item*
- In the Add New Item window, by default web form is selected, leave it selected

- In the file name box, change the file name to contact.aspx, check the box *Place code in separate file* and then left click *Select*

You will notice when the editor created our file, it created two:

- Contact.aspx
- Contact.aspx.cs

The first file contains our presentational logic, such as HTML, CSS, JavaScript, our web and validation controls, while the second file contains our business logic such as event handlers that will send the email message to our specified recipient, and perform client and server-side validation.

Working with our presentation file (contact.aspx) first

Even though these files are associated together, it often helps to work with them separately, and more importantly, work with the presentational file first, and then focus on making the form do what we want later. For the purposes of this article, we'll be working strictly from code view, as that's the best way to learn how the environment works.

Creating and analyzing the form

Inside the file, you'll notice that it looks just like any other HTML page. It has a normal HTML structure that you would expect, but behind this ordinary markup, there's a wealth of power and flexibility we'll tap into. You'll notice that by default our editor created a <form> tag. By default, all ASP.NET web forms have a form tag and we'll understand why that's important later in the article. Inside the opening and closing form tag, let's create a basic table structure that will collect first and last name, email, and comments from our visitor as shown below:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="contact.aspx.cs"
Inherits="contact" %>
<!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></title>
</head>
<body>
<form id="form1" runat="server">
<h3>My Contact Form</h3>
<table id="contact" cellspacing="0">
<tr>
<td>First Name:</td>
<td></td>
</tr>
<tr>
<td>Last Name:</td>
<td></td>
</tr>
<tr>
<td>Email:</td>
```

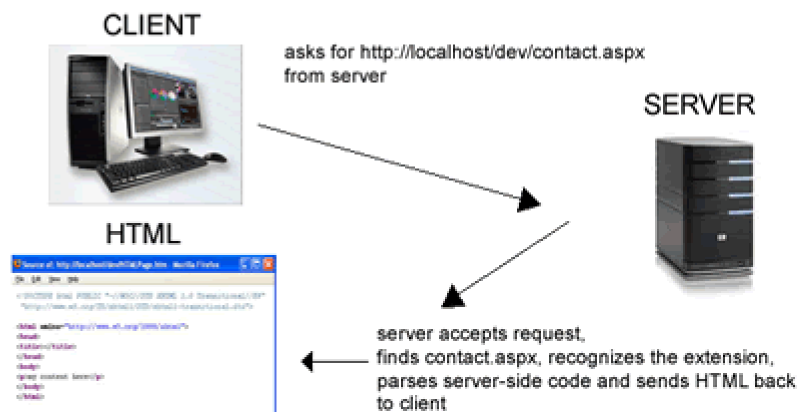
```
<td></td>
</tr>
<tr>
<td>Comments:</td>
<td></td>
</tr>
</table>
</form>
</body>
</html>
```

As you can see from the example above, we're dealing with an ordinary table with an ID that we can use for controlling the presentational aspect of our page, that is, a style sheet. However, for the purposes of this article, we'll focus on the ASP.NET environment in relation to the form.

Client-server architecture demystified

In the web environment, everything is initiated by using protocols, requests for services or a combination of both. Specifically, when working with server-side technologies on the web developers work in client-server architectures. That is, a client such as a local computer requests through an HTTP protocol that a file be sent to it from a server. In response, a server receives the request, accepts it, and finds the file and returns the file to the client.

In some website environments, the only responsibility the server has is accepting the request, and delivering the file to the client. In other website environments, when dynamic pages exist, such as pages that end in ASP, ASPX, the server accepts the request, notices the page has an ASP extension, and as a result, knows that it needs to read, interpret and parse any server-side code before sending the file back to the client with the parsed HTML included. For a little added clarity, the illustration below clarifies the process.



At this point, you may be wondering how the server knows what to do with these special files? It all boils down to Internet Services Application Program Interface (ISAPI). Every web server has the ability for files to interface with it, specifically by installing dynamic link libraries (DLL) that contains code

functions that instruct the web server to interpret the request, and if the file has certain extensions, then parse the server-side code before sending the result back to the browser.

Understanding web controls

If you're coming from the traditional ASP or PHP environment, the way you're used to developing form fields is vastly different in ASP.NET. Instead of using <input> tags, we use web controls. When Microsoft created ASP.NET, it basically reinvented web controls that run on the server-side, and then sends back an HTML equivalent. For example, a text box control is a web control, when parsed by the server, generates an HTML input tag. When the generated HTML comes back from the server, the ASP.NET engine includes special values for each control, such as the name attribute which comes from the ID of the control. We'll see more on that shortly. The most common types of web controls include:

- Text boxes
- Radio buttons
- Drop down lists
- Check boxes

In order to create our ordinary input fields, we'll be using text box web controls. Let's add three of them to our file by using the code below:

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="contact.aspx.cs"
Inherits="contact" %>
<!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></title>
</head>
<body>
<form id="form1" runat="server">
<h3>My Contact Form</h3>
<table id="contact" cellspacing="0">
<tr>
<td>First Name:</td>
<td><asp:TextBox ID="FNameTB" runat="server" /></td>
</tr>
<tr>
<td>Last Name:</td>
<td><asp:TextBox ID="LNameTB" runat="server" /></td>
</tr>
<tr>
<td>Email:</td>
<td><asp:TextBox ID="EmailTB" runat="server" /></td>
</tr>
<tr>
<td>Comments:</td>
<td></td>
</tr>
```

```
</table>
</form>
</body>
 </html>
```

As you can see from the example above, our web control(s) look a bit odd, but really they're simple. Using the first one as an example, let's analyze it:

```
<td><asp:TextBox ID="FNameTB" runat="server" /></td>
```

We nest the web control inside a table cell. The web control parts are:

- <asp:TextBox
    - Instructs the ASP.NET engine to parse this control as in input field in HTML
- ID=FNameTB
    - Each web control must have a unique name. This also serves the purpose of accessing the controls properties and methods in our code-behind file
- Runat=server
    - Instructs the control to be run at the server

We purposely left the comments field alone for the moment, but we'll get to it in a moment.

Open the Web site in a browser

In order to see the results of our work, open your preferred web browser and type the following:

http://localhost/dev/contact.aspx

Don't be concerned if the page doesn't load immediately. Since the ASP.NET environment compiles each web form into Dynamic Link Libraries (DLL), the first time you load a web form into a browser; it has to compile any code or control before it can render the page. However, as a result, since it's compiled, if there are no changes, subsequent requests for the page should speed up and no delay should be evident.

View Source

Once your web page has rendered, to reiterate what the ASP.NET environment is doing, view the source of the page in your browser and you should see this:

```
<input type="hidden" name="__VIEWSTATE"
      <input type="hidden" name="__EVENTVALIDATION"
</div>
<h3 align="center">My Contact Form</h3>
<table id="contact" cellspacing="0" align="center">
<tr>
<td>First Name:</td>
<td><input name="FNameTB" type="text" id="FNameTB" />
```

```
<asp:TextBox ID="FNameTB" runat="server" />
```

As you can see from the example above, the ASP.NET environment has automatically replaced the text box web controls with ordinary HTML input tags for first and last name, as well as email, and has used the ID of the control as the value for the name attribute. Keep note of the view state tag circled in red above, we'll get to that later in the article. Since a comments field is generally thought to contain more text than an input tag, we use a text area tag instead. Let's go ahead and create the text area tag by modifying our file as shown below:

```
<tr>
<td>Comments:</td>
<td><asp:TextBox ID="CommentsTB" runat="server" TextMode="MultiLine" /></td>
</tr>
```

As you can see from the example above, in order to get a <textarea> tag, we use the text mode property of our text box control and set its value to multiline. Save your file and refresh your page to see the results.

Create our submit and reset buttons

In order for our contact form to send an email message to a specified recipient, we need to add buttons to our form that will trigger code to make this happen. Modify our file as shown below:

```
<tr>
<td colspan="2"><asp:Button ID="btnSubmit" runat="server" Text="Submit"
OnClick="SendMail"/><asp:Button ID="btnReset" runat="server" Text="Reset"
OnClick="Reset"/></td>
</tr>
```

As you can see from the example above, we added an extra table row, and created two web control buttons, one for sending the mail, and another one for clearing the form fields. One difference in these buttons from the others includes using the text property to set their text values to submit and reset accordingly that will show a textual description for the buttons. Another important difference is the

onclick property. The first confusing aspect of this property is one tends to think this is a client-side event, in this case, it's not, it's a server-side event that will trigger a server-side event to occur in our code-behind file that will ensure validation takes place, as well as emailing the results of information to our specified recipient, which we'll get to next. Save your changes and close the file. Open the code-behind file (contact.aspx.cs) and add the following event handlers below the page load event handler:

```
protected void Page_Load(object sender, EventArgs e)
    {

    }

protected void SendMail(object sender, EventArgs e)
    {

}

protected void Reset(object s, EventArgs e)
    {

}
```

As you can see from the examples above, the name of our event handler matches our *onclick* event in our presentational file. It should be noted that if you set an *onclick* event on a button without creating a corresponding event handler, your web form will throw an error. We'll discuss the code within this file shortly.

Event driven environment

In typical web environments, such as PHP or ASP, when you want an action to occur, like a form submission that sends an email, you create an input button and specify its type as submit. When the button is clicked, the file that's attached to the action attribute of the form, typically a script is then executed performing the business logic. Even though an "event" is technically occurring, it's typically not considered an event, since that's the default behavior of a form.

When Microsoft created the .NET framework, and integrated ASP into it, it created the ability for any web form to respond to events, such as a button click. As a result, since our submit button has an *onclick* event declared, when the button is "clicked", it causes an "event" to execute on the server, thus forcing our presentation file to look for the associated code behind file, locate our "event", and execute the business logic.

Examining the code

Before creating and examining our code, it's important to understand that our code-behind file is being written in C# (pronounced C-sharp). C# was created by Microsoft in an attempt to compete with Sun Microsystems Java, both of which are object-oriented languages. The idea behind C# was to leverage the power of the C constructs, while leveraging the easy syntax of Visual Basic. As a result, all code-behind files for ASP.NET are written in object-orientated languages, which are then compiled by the server into efficient binary or dynamic link libraries.

Continuing, you'll see the following:

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;

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

    }
protected void SendMail(object sender, EventArgs e)
    {

    }
protected void Reset(object s, EventArgs e)
    {

    }
}
```

Starting from the beginning of the file, you'll notice many *using* statements. Since ASP was integrated in the .NET framework, it has access to all the classes and libraries; you just have to import them. In C#, you give your web form access to them by declaring *using* statements, which imports which classes you want your web form to have. The only one we need to add is the class for email, which is found in the *System.Net.Mail* namespace. Simply modify the code to include an additional *using* statement as shown below:

```
using System;
using System.Collections.Generic;
using System.Web;
using System.Web.UI;
using System.Web.UI.WebControls;
using System.Net.Mail;
```

Understanding event handlers

As we continue through the code, let's examine the event handlers:

- Protected: means any variables, or actions inside this event are protected from the rest of the application and aren't in scope to the rest of the page.
- Void: means the event handler doesn't return a value.
- Object sender: contains information about the event that's been triggered or executed
- EventArgs e: contains information about the control that triggered the event to execute

Extending the Send Mail Event

In order for us to collect the information entered in our form and send the mail to our specified recipient, we'll create a mail message object. We do this by adding the code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();

    }
```

As you can see from the example above, we want to create our object from the mail message class. At this point, in object-oriented languages, when you assign a variable with the new keyword, followed by a class constructor, you are said to be creating an instance of an object of that class. As a result, our object, mail, has access to all properties and methods of the mail message class.

Setting the from address of our mail object

When we send the mail to the specified recipient, we want to know who contacted us so that we can follow up with them at our convenience. Since we have an email text box that's used to collect our visitor's email address, we'll use that as a way to touch base with our client when the need arises. Let's modify our code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        mail.From = new MailAddress(EmailTB.Text);
    }
```

As you can see from the example above, we set the property, from, of our mail object to a new instance of the mail address class, and pass in our email text box control as an argument.

Setting the email recipient of our mail object

In order for our clients to reach us, we need to send the email result to a specified recipient. We do this by adding the following code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        mail.From = new MailAddress(EmailTB.Text);
        mail.To.Add("myemail@domain.com");
    }
```

As you can see from the example above, we first set the property, to, of our mail object that specifies a collection, that is, a collection of email address(s) that this message could be sent to. For example, you could send this email to other recipients. Then we use the add method to add the specified email address to our mail object's collection and pass in a literal string of an appropriate email address.

Setting the subject line of our mail object

In most cases, we'll want to have a pre-defined subject line so that when our clients do email us, we'll be easily able to identify what the message is about. To do this, we modify our code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        mail.From = new MailAddress(EmailTB.Text);
        mail.To.Add("myemail@domain.com");
        mail.Subject = "Contact Us";
    }
```

As you can see from the example above, we set the subject property of our mail object to a literal string of our choice.

Setting the content type of our mail object

When sending an email message from a contact form, you have the option of specifying whether it will be plain text or HTML. Some people choose not to accept HTML encoded emails because they can contain malicious code that can be used to capture sensitive information or spread viruses and/or worms on a person's computer. However, since we're sending this email to ourselves, we know the content will be legitimate; therefore, we'll go ahead and specify that this email can contain HTML by modifying the code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        mail.From = new MailAddress(EmailTB.Text);
        mail.To.Add("myemail@domain.com");
        mail.Subject = "Contact Us";
        mail.IsBodyHtml = true;
    }
```

As you can see from the example above, we set the IsBodyHTML property of our mail object to true, which will allow any HTML enabled email client to parse and read the HTML.

Collecting the forms contents through our mail object

In order for us to collect the information that our visitors entered in our form, we need to set our mail object's body property to a literal string that will ultimately contain a concatenated string from our form in a nicely formatted message that's readable through an email client. Concatenation is a fancy word for saying *"append to an existing string"*. Let's add this functionality by adding the code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        mail.From = new MailAddress(EmailTB.Text);
        mail.To.Add("myemail@domain.com");
        mail.Subject = "Contact Us";
        mail.IsBodyHtml = true;
        mail.Body = "First Name:  " + FNameTB.Text + "<br />";
        mail.Body += "Last Name:  " + LNameTB.Text + "<br />";
        mail.Body += "Email:  " + EmailTB.Text + "<br />";
        mail.Body += "Comments:  " + CommentsTB.Text + "<br />";
    }
```

As you can see from the example above, we first set the body property of our mail object to a literal string that contains "First Name:", then we concatenate by using the plus (+) sign the value entered by our visitor via our text box control by using its text property, and then to create separation for the next line in our string, we append an ordinary HTML break tag. We continue the body of the mail message by concatenating to the existing string our additional form fields, followed by their appropriate controls.

Sending the email through SMTP client class

Since we have set everything we need, we now need to send the mail message to our recipient. In order for us to do this we need to create a new instance of an object based off the SMTP client class. Before moving forward, it should be noted that all our script can do is send the email message to a relay server and hope the server delivers the message. Once our web form sends the email message, all responsibility for sending the message is delegated to the relay server, in our case, IIS. With that said, let's modify our page with the code as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        MailMessage mail = new MailMessage();
        mail.From = new MailAddress(EmailTB.Text);
        mail.To.Add("myemail@domain.com");
        mail.Subject = "Contact Us";
        mail.IsBodyHtml = true;
        mail.Body = "First Name:  " + FNameTB.Text + "<br />";
        mail.Body += "Last Name:  " + LNameTB.Text + "<br />";
        mail.Body += "Email:  " + EmailTB.Text + "<br />";
        mail.Body += "Comments:  " + CommentsTB.Text + "<br />";

        SmtpClient smtp = new SmtpClient();
        smtp.Host = "your_relay_mail_server";
        smtp.Send(mail);
    }
```

As you can see from the example above, we create a new instance of the SMTP client class with an object named *SMTP*. From there, we set its host property to a relay server. It should be noted that the way of sending email to a relay server on a production server can differ significantly between hosts. The above method is meant to be a generic look at how to do it. As a result, if the above doesn't work for you, ask your hosting provider for additional details. Lastly, to send our results from our form, we set the send

method of our SMTP object and pass our mail object as an argument. Save your file. We'll send a test email a little later on.

Creating Placeholder controls

Before moving on, we need to add two place holder controls to our page. Right now, if we were to submit our form, other than an email being delivered to our inbox, there would be no visual indication of a successful submission. As a result, we need to provide a message to our visitor's informing them of a successfully sent email.

We have two options: (1) after sending the email, use a redirect to send our visitors to a separate success page, or (2) create place holder controls that will show or hide content based on success or failure. Generally, for the sake of maintenance, it's wise to keep everything in one file. In order to create these controls, open contact.aspx and add the following controls as shown below:

```
<body>
<asp:PlaceHolder ID="formPH" runat="server" Visible="true">
<form id="form1" runat="server">

…more code

</form>
</asp:PlaceHolder>

<asp:PlaceHolder ID="sucessPH" runat="server" Visible="false">
<p>Thank you for your submission.</p>
</asp:PlaceHolder>
</body>
```

As you can see from the example above, right after our opening body tag, we create a place holder control named *formPH* and set it's visibility to true. Since it has the contents of our form, we want to show it initially. Then, right before the closing body tag, we add another place holder control named *successPH* and set its visibility to false. Continuing, open contact.aspx.cs and add the following code as shown below:

```
smtp.Send(mail);
formPH.Visible = false;
sucessPH.Visible = true;
```

As you can see from the example above, right after sending our email message, we set our form's place holder control's visibility to false, and instead show our success message by setting its visibility to true. Save your files.

Extending the reset event handler

Right now, there's no way to clear our form fields of existing data. Let's fix this by adding the code shown below to our existing reset event handler:

```
protected void Reset(object s, EventArgs e)
    {
        FNameTB.Text = "";
        LNameTB.Text = "";
        EmailTB.Text = "";
        CommentsTB.Text = "";
    }
```

As you can see from the example above, we set each web control's text property to an empty string. Save your file, refresh your browser; fill in some fields and press reset.

Validation

Since we have created our form, and it's sending the information we need, we need to implement functionality that ensure our fields are filled in before sending the email. Generally, when you provide forms for your visitors to fill out, you want a certain amount of information entered before it's allowed to submit the email to the specified recipient. Validation can be done one of two ways or both, just depends on how much you want to foolproof your form.

The first way is to use client-side validation using JavaScript, which is only performed client-side, in other words, only executed on the visitor's computer. This is often done to provide instant feedback to the visitor as to any errors that occurred. There's only one disadvantage to relying on this approach: if JavaScript is disabled in a visitor's browser, then validation fails to execute, leaving your web form with empty details that you may require.

The second way is use server-side validation using the behaviors present within your server-side language. This method is far more reliable because even if a visitor has JavaScript disabled in a browser, server-side validation scripts can still execute from the server and provide feedback as to any errors that occurred.

Generally, it's recommended to use a combination of both, that is, using JavaScript to ensure instant feedback that limits post backs to the server, and then using server-side scripts to check sensitive fields to ensure that the visitor did in fact give you the information you need.

Creating required fields in ASP.NET

Generally, in PHP and ASP, you create a separate JavaScript file that contains functions to check fields for empty values. Then, within your page, usually at the top, you would provide server-side validation to ensure those same fields do in fact contain a value. In essence, you have two files. In the ASP.NET environment, they made this process much easier and a bit more intuitive. When Microsoft created ASP.NET, they decided to package it with an additional set of controls that are effectively grouped together in a set of validation controls. Out of this group and arguably the easiest to use are required field validators, which do exactly what they are called: they require that certain fields are filled in before

allowing the form to submit, more importantly however, they produce the JavaScript automatically for us. Let's go back to contact.aspx and modify our code as shown below:

```
<table id="contact" cellspacing="0">
<tr>
<td>First Name:</td>
<td><asp:TextBox ID="FNameTB" runat="server" />
<asp:RequiredFieldValidator ID="rfvFName" runat="server"
ControlToValidate="FNameTB" ErrorMessage="First Name is required"
Display="Dynamic" />
</td>
</tr>
<tr>
<td>Last Name:</td>
<td><asp:TextBox ID="LNameTB" runat="server" />
<asp:RequiredFieldValidator ID="rfvLName" runat="server"
ControlToValidate="LNameTB" ErrorMessage="Last Name is required"
Display="Dynamic" />
</td>
</tr>
<tr>
<td>Email:</td>
<td><asp:TextBox ID="EmailTB" runat="server" />
<asp:RequiredFieldValidator ID="rfvEmail" runat="server"
ControlToValidate="EmailTB" ErrorMessage="Email is required"
Display="Dynamic" />
</td>
</tr>
<tr>
<td>Comments:</td>
<td><asp:TextBox ID="CommentsTB" runat="server" TextMode="MultiLine" />
<asp:RequiredFieldValidator ID="rfvComments" runat="server"
ControlToValidate="CommentsTB" ErrorMessage="Comments are required"
Display="Dynamic" />
</td>
```

As you can see from the example above, we have used required field validators to validate each text box. The validation controls parts are:

- ID: Each control must have a unique ID which allows us to identify it if the need arises
- Runat=server: Instructs this control to be run by the server
- ControlToValidate: Instructs our validation controls which control (s) we want to validate, in our case, each corresponding text box control
- ErrorMessage: Provides error messages to our visitors informing them of the errors present
- Display=Dynamic: Is set to ensure the error messages would not be in source code until they are executed.

After saving your file with the new controls, refresh your page in your browser, leave all fields empty and click the submit button, you should have four error messages as shown below:

My Contact Form

First Name: First Name is required
Last Name: Last Name is required
Email: Email is required
Comments: Comments are required
Submit Reset

It should be noted the placement of these controls is essential. The error messages for the controls will show wherever the control is placed. Since we placed the validation controls right before the closing table cell for each web control, the error messages are shown right after the controls.

You don’t see error messages?

If your error messages don’t show, it likely means that you’re missing the validation JavaScript file in the default installation of IIS, which is at the following directory location:

C:\Inetpub\wwwroot\aspnet_client\system_web\1_1_4322

The file is appropriately named WebUIValidation.js. To fix this issue, run the following command from the command prompt:

C: \WINDOWS\Microsoft.NET\Framework\[version]\ aspnet_regiis.exe -c

It should be noted that by default, all virtual directories point to the local installation of IIS. As a result, the ability for your virtual directory to see this file shouldn’t be an issue, unless of course, it’s not there.

Enabling server-side validation

Even though we have established client-side validation, if JavaScript is disabled in a visitor’s browser, the validation controls that produce JavaScript will fail. Fortunately, ASP.NET makes the server-side check quite easy to implement. Switch back to your code-behind file and modify the send mail event to the following as shown below:

```
protected void SendMail(object sender, EventArgs e)
    {
        if (!IsValid)
        {
            return;
        }
        else
        {
            MailMessage mail = new MailMessage();
            mail.From = new MailAddress(EmailTB.Text);
            mail.To.Add("myemail@domain.com");
            mail.Subject = "Contact Us";
            mail.IsBodyHtml = true;
            mail.Body = "First Name: " + FNameTB.Text + "<br />";
            mail.Body += "Last Name: " + LNameTB.Text + "<br />";
            mail.Body += "Email: " + EmailTB.Text + "<br />";
            mail.Body += "Comments: " + CommentsTB.Text + "<br />";


            SmtpClient smtp = new SmtpClient();
            smtp.Host = "your_relay_server";
            smtp.Send(mail);
        }
    }
```

As you can see from the example above, we perform a conditional check within our send mail event. Inside the check, we perform a Boolean check on the IsValid property, which essentially checks the values of all our controls in our page. As a result, if any of the controls come back as false, meaning they're empty, we issue a return statement, that stops the rest of the event from firing and provides our visitor with an error message as to what went wrong. If all controls that we require come back as true, meaning they're filled in, then the rest of our send mail event is allowed to execute, and subsequently, send our email message. Save your file, disable JavaScript and leave a few fields blank, and the error messages should show. At this point, we have a completed file that is ready and working.

Publishing your file

When publishing your file to your remote server, make sure to create an application where this form can reside and change the relay server where appropriate.

Sending a test email

Presuming the SMTP client is configured correctly on the remove server, go ahead and fill in our fields, click submit, and you should get an email from our form delivered to your inbox in a few minutes.

View State

One more important concept should be mentioned before finishing the article. By default, the web is a stateless environment. In essence, when you visit a website and navigate between different pages, the website has no idea what page you last visited. Often times, when writing websites that have dynamic activity such as displaying categories, and their associated products, web programmers will use a query string so that a web page can remember where it came from, and then display the associated content.

In our case, when visitors submit form fields with validation enforced, if errors occur, the page post backs to the server and any existing information in the fields is lost. In PHP or ASP, the only way around this was to create a variable for each form field, and set it to the request value of the form field, and then evaluate the variable value out, which is a pain.

With ASP.NET, they have resolved this issue by a concept called view state. In essence, view state is enabled when you use a form tag, which if you remember from earlier in the article, form tags are created by default on all web forms. More importantly view state is a hidden input tag with a special value used by the ASP.NET engine that keeps the values of our web controls stored in memory. This is important when you enforce validation on your web forms. Instead of creating additional variables, ASP.NET does the hard work for us. If you visit our file in the browser, fill in a couple fields, but leave the other two empty, press submit, you'll notice when the page posts back to the server, the existing information in the first two fields is still there, done automatically by ASP.NET!

However it should be noted that view state doesn't solve the problem of websites remembering the last page visited through a stateless environment, such as the query string method mentioned in our earlier category to products example. Also, it should be noted that additional web controls produce a larger view state, which can affect performance, something to keep in mind as you continue your work with web controls.

Summary

In this article you learned how to create a contact form using ASP.NET. Additionally, you learned the following:

- How to download and install:
  - .NET framework
  - Visual Web Developer Express
- How to install a local web server, Internet Information Services (IIS)
- The .NET framework and the integration of ASP.NET
- How to create a web form and its associated code-behind file
- Within ASP.NET environment:
  - Web controls, such as text boxes and their associated properties and values
  - How to create event driven methods
  - Validation controls that make client and server-side validation a breeze
  - View state and it's relation to the stateless web environment

From the knowledge gained in this article, one should be able to take this simple web form, identify the information needed, be able to validate form fields, as well as send information collected to a specified email address. As a result, this web form could be modified to meet the needs of any web application for a business, company or organization to meet the needs of their customers, keep needless email from spamming our system, in a more intuitive, reliable and secure method.

If you have questions, please follow the link below:

http://midwestwebdesign.net/tutorials/contact.aspx