ASP.NET - UNIT 3

Validation Controls:

Various Controls like Require Field, Compare,

Range, RegularExpression ,

Custom, Validation Summary,

Dynamic controls.

Debugging ASP.NET pages: Error Handling: Custom Error Page,

Using Debugging Tools: Debugger and Trace Facility.

Validation Controls:-

ASP.NET validation controls validate the user input data to ensure that useless, unauthenticated, or contradictory data don't get stored. *ASP.NET provides a set of validation controls that provide an easy-to-use but powerful way to check for errors and, if necessary, display messages to the user.*

ASP.NET provides the following validation controls:

RequiredFieldValidator

RangeValidator

CompareValidator

RegularExpressionValidator

CustomValidator

ValidationSummary

BaseValidator Class

The validation control classes are inherited from the BaseValidator class hence they inherit its properties and methods. Therefore, it would help to take a look at the properties and the methods of this base class, which are common for all the validation controls:

Members	Description
ControlToValidate	Indicates the input control to validate.
Display	Indicates how the error message is shown.
EnableClientScript	Indicates whether client side validation will take.
Enabled	Enables or disables the validator.
ErrorMessage	Indicates error string.
Text	Error text to be shown if validation fails.
IsValid	Indicates whether the value of the control is valid.
SetFocusOnError	It indicates whether in case of an invalid control, the focus should switch to the related input control.
ValidationGroup	The logical group of multiple validators, where this control belongs.
Validate()	This method revalidates the control and updates the IsValid property.

RequiredFieldValidator Control

The RequiredFieldValidator control ensures that the required field is not empty. It is generally tied to a text box to force input into the text box.

The syntax of the control is as given:

```
<asp:RequiredFieldValidator ID="rfvcandidate"
   runat="server" ControlToValidate ="ddlcandidate"
   ErrorMessage="Please choose a candidate"
   InitialValue="Please choose a candidate">
</asp:RequiredFieldValidator>
```

RangeValidator Control

The RangeValidator control verifies that the input value falls within a predetermined range.

It has three specific properties:

Properties	Description
Type	It defines the type of the data. The available values are: Currency, Date, Double, Integer, and String.
MinimumValue	It specifies the minimum value of the range.
MaximumValue	It specifies the maximum value of the range.

The syntax of the control is as given:

```
<asp:RangeValidator ID="rvclass" runat="server"
ControlToValidate="txtclass"
   ErrorMessage="Enter your class (6 - 12)" MaximumValue="12"
   MinimumValue="6" Type="Integer">
```

```
</asp:RangeValidator>
```

CompareValidator Control

The CompareValidator control compares a value in one control with a fixed value or a value in another control.

It has the following specific properties:

Properties	Description
Type	It specifies the data type.
ControlToCompare	It specifies the value of the input control to compare with.
ValueToCompare	It specifies the constant value to compare with.
Operator	It specifies the comparison operator, the available values are: Equal, NotEqual, GreaterThan, GreaterThanEqual, LessThan, LessThanEqual, and DataTypeCheck.

The basic syntax of the control is as follows:

```
<asp:CompareValidator ID="CompareValidator1" runat="server"
   ErrorMessage="CompareValidator">


</asp:CompareValidator>
```

RegularExpressionValidator

The RegularExpressionValidator allows validating the input text by matching against a pattern of a regular expression. The regular expression is set in the ValidationExpression property.

The following table summarizes the commonly used syntax constructs for regular expressions:

Character Escapes	Description
\b	Matches a backspace.
\t	Matches a tab.
\r	Matches a carriage return.
\v	Matches a vertical tab.
\f	Matches a form feed.
\n	Matches a new line.
\	Escape character.

Apart from single character match, a class of characters could be specified that can be matched, called the metacharacters.

Metacharacters	Description
.	Matches any character except \n.
[abcd]	Matches any character in the set.
[^abcd]	Excludes any character in the set.
[2-7a-mA-M]	Matches any character specified in the range.

\w	Matches any alphanumeric character and underscore.
\W	Matches any non-word character.
\s	Matches whitespace characters like, space, tab, new line etc.
\S	Matches any non-whitespace character.
\d	Matches any decimal character.
\D	Matches any non-decimal character.

Quantifiers could be added to specify number of times a character could appear.

Quantifier	Description
*	Zero or more matches.
+	One or more matches.
?	Zero or one matches.
{N}	N matches.
{N,}	N or more matches.
{N,M}	Between N and M matches.

The syntax of the control is as given:

```
<asp:RegularExpressionValidator ID="string" runat="server"
ErrorMessage="string"
   ValidationExpression="string" ValidationGroup="string">
```

```
</asp:RegularExpressionValidator>
```

CustomValidator

The CustomValidator control allows writing application specific custom validation routines for both the client side and the server side validation.

The client side validation is accomplished through the ClientValidationFunction property. The client side validation routine should be written in a scripting language, such as JavaScript or VBScript, which the browser can understand.

The server side validation routine must be called from the control's ServerValidate event handler. The server side validation routine should be written in any .Net language, like C# or VB.Net.

The basic syntax for the control is as given:

```
<asp:CustomValidator ID="CustomValidator1" runat="server"

   ClientValidationFunction=.cvf_func.
ErrorMessage="CustomValidator">

</asp:CustomValidator>
```

ValidationSummary

The ValidationSummary control does not perform any validation but shows a summary of all errors in the page. The summary displays the values of the ErrorMessage property of all validation controls that failed validation.

The following two mutually inclusive properties list out the error message:

ShowSummary : shows the error messages in specified format.

ShowMessageBox : shows the error messages in a separate window.

The syntax for the control is as given:

```
<asp:ValidationSummary ID="ValidationSummary1" runat="server"
   DisplayMode = "BulletList" ShowSummary = "true"
HeaderText="Errors:" />
```

Validation Groups

Complex pages have different groups of information provided in different panels. In such situation, a need might arise for performing validation separately for separate group. This kind of situation is handled using validation groups.

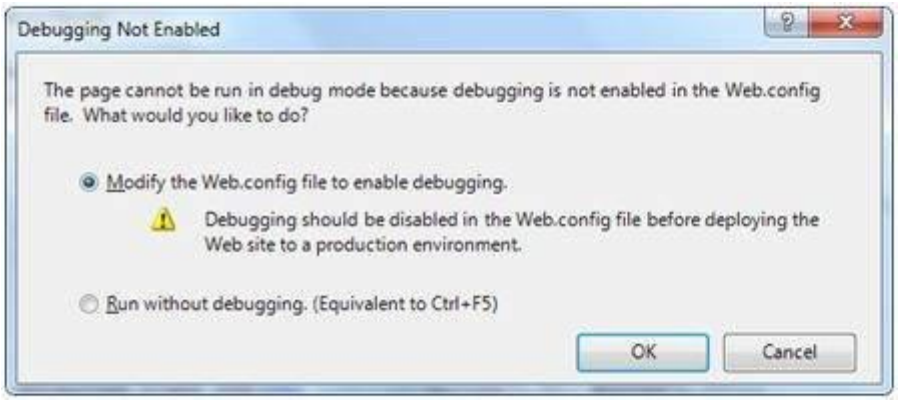
To create a validation group, you should put the input controls and the validation controls into the same logical group by setting their ValidationGroup property.

Debugging ASP.NET pages

Debugging allows the developers to see how the code works in a step-by-step manner, how the values of the variables change, how the objects are created and destroyed, etc.

Debugging is the process of adding breakpoints to an application. These breakpoints are used to pause the execution of a running program. This allows the developer to understand what is happening in a program at a particular point in time.

When the site is executed for the first time, Visual Studio displays a prompt asking whether it should be enabled for debugging:


Debugging Not Enabled

The page cannot be run in debug mode because debugging is not enabled in the Web.config file. What would you like to do?

● Modify the Web.config file to enable debugging.

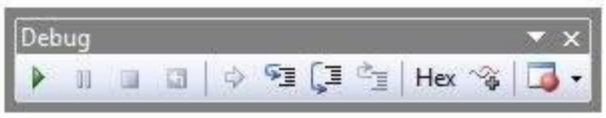
Debugging should be disabled in the Web.config file before deploying the Web site to a production environment.

○ Run without debugging. (Equivalent to Ctrl+F5)

When debugging is enabled, the following lines of codes are shown in the web.config:

```
<system.web>
   <compilation debug="true">
      <assemblies>
      ..............
      </assemblies>
   </compilation>
</system.web>
```

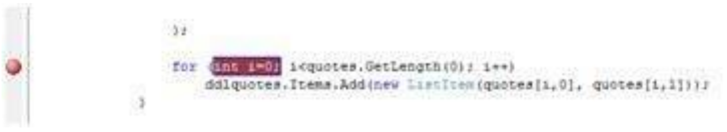
The Debug toolbar provides all the tools available for debugging:



Breakpoints

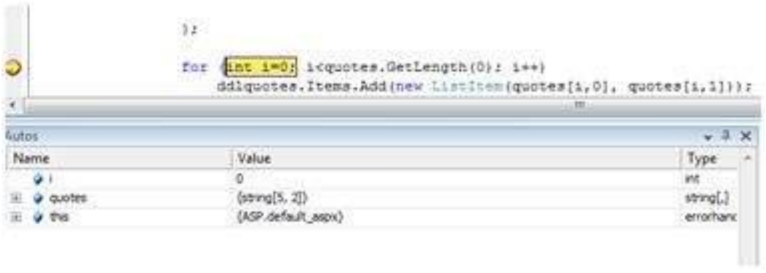
Breakpoints specifies the runtime to run a specific line of code and then stop execution so that the code could be examined and perform various debugging jobs such as, changing the value of the variables, step through the codes, moving in and out of functions and methods etc.

To set a breakpoint, right click on the code and choose insert break point. A red dot appears on the left margin and the line of code is highlighted as shown:



```
            };

            for (int i=0; i<quotes.GetLength(0); i++)
                ddlquotes.Items.Add(new ListItem(quotes[i,0], quotes[i,1]));
        }
```

Next when you execute the code, you can observe its behavior.



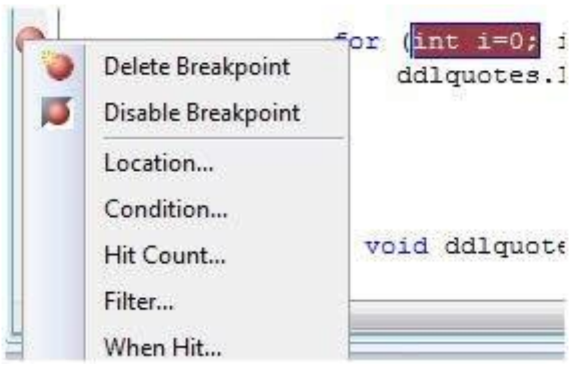
```
            };

            for (int i=0; i<quotes.GetLength(0); i++)
                ddlquotes.Items.Add(new ListItem(quotes[i,0], quotes[i,1]));
```

Autos

Name	Value	Type
i	0	int
quotes	{string[5, 2]}	string[,]
this	{ASP.default_aspx}	errorhanc

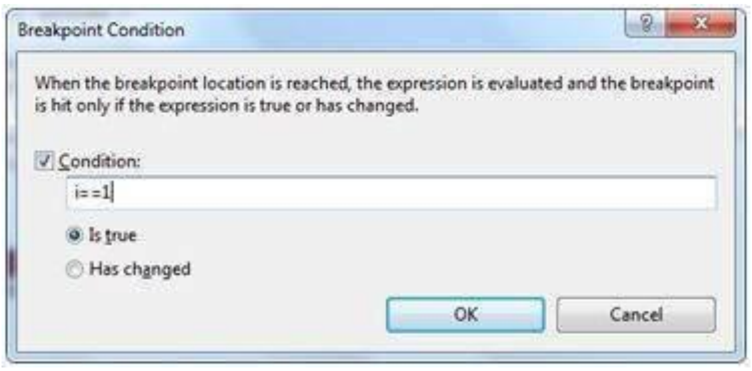
At this stage, you can step through the code, observe the execution flow and examine the value of the variables, properties, objects, etc.

You can modify the properties of the breakpoint from the Properties menu obtained by right clicking the breakpoint glyph:



```
for (int i=0; i
    ddlquotes.I

void ddlquote
```

The location dialog box shows the location of the file, line number and the character number of the selected code. The condition menu item allows you to enter a valid expression, which is evaluated when the program execution reaches the breakpoint:



Breakpoint Condition

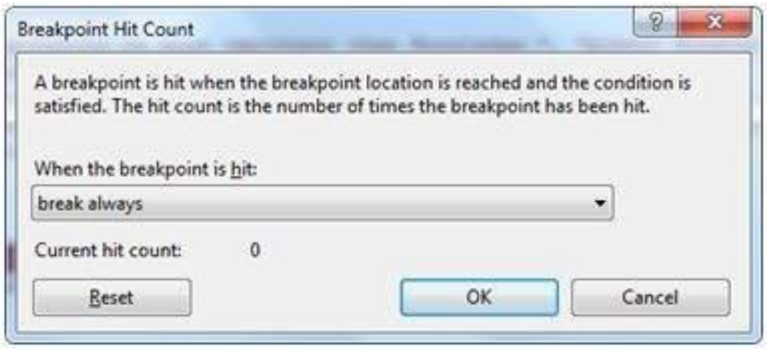
When the breakpoint location is reached, the expression is evaluated and the breakpoint is hit only if the expression is true or has changed.

[✓] Condition:

i==1

[● Is true]
○ Has changed

The Hit Count menu item displays a dialog box that shows the number of times the break point has been executed.

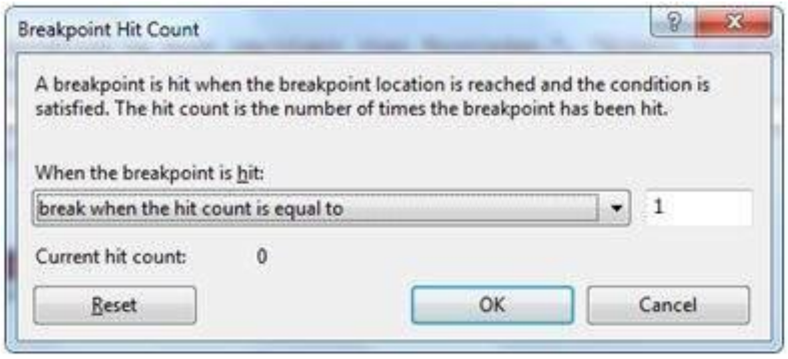

Breakpoint Hit Count

A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

When the breakpoint is hit: break always

Current hit count: 0

Clicking on any option presented by the drop down list opens an edit field where a target hit count is entered. This is particularly helpful in analyzing loop constructs in code.

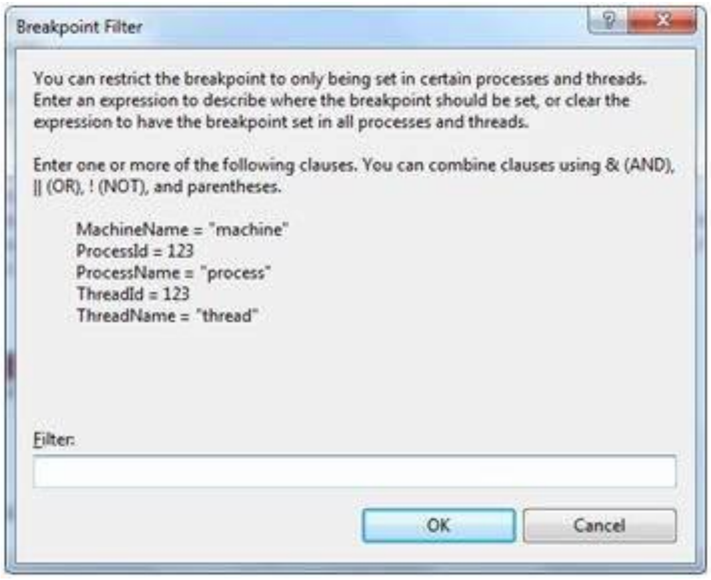

Breakpoint Hit Count

A breakpoint is hit when the breakpoint location is reached and the condition is satisfied. The hit count is the number of times the breakpoint has been hit.

When the breakpoint is hit: break when the hit count is equal to 1

Current hit count: 0

The Filter menu item allows setting a filter for specifying machines, processes, or threads or any combination, for which the breakpoint will be effective.


Breakpoint Filter

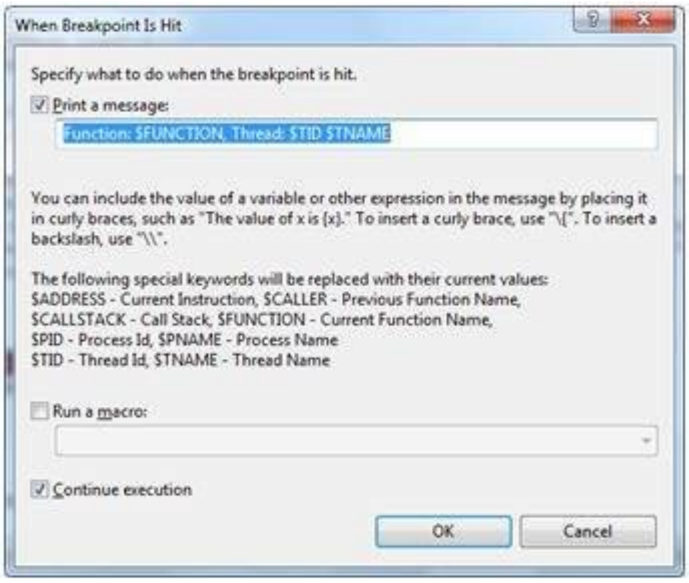
You can restrict the breakpoint to only being set in certain processes and threads. Enter an expression to describe where the breakpoint should be set, or clear the expression to have the breakpoint set in all processes and threads.

Enter one or more of the following clauses. You can combine clauses using & (AND), || (OR), ! (NOT), and parentheses.

```
MachineName = "machine"
ProcessId = 123
ProcessName = "process"
ThreadId = 123
ThreadName = "thread"
```

Filter:

The When Hit menu item allows you to specify what to do when the break point is hit.


When Breakpoint Is Hit

Specify what to do when the breakpoint is hit.

[✓] Print a message:

Function: $FUNCTION, Thread: $TID $TNAME

You can include the value of a variable or other expression in the message by placing it in curly braces, such as "The value of x is {x}." To insert a curly brace, use "\{". To insert a backslash, use "\\".

The following special keywords will be replaced with their current values:
$ADDRESS - Current Instruction, $CALLER - Previous Function Name,
$CALLSTACK - Call Stack, $FUNCTION - Current Function Name,
$PID - Process Id, $PNAME - Process Name
$TID - Thread Id, $TNAME - Thread Name

[ ] Run a macro:

[✓] Continue execution

The Debug Windows

Visual Studio provides the following debug windows, each of which shows some program information. The following table lists the windows:

Window	Description
Immediate	Displays variables and expressions.
Autos	Displays all variables in the current and previous statements.
Locals	Displays all variables in the current context.
Watch	Displays up to four different sets of variables.
Call Stack	Displays all methods in the call stack.
Threads	Displays and control threads.

<u>Error handling in ASP.NET has three aspects:</u>

- **Tracing -** tracing the program execution at page level or application level.
- **Error handling -** handling standard errors or custom errors at page level or application level.
- **Debugging -** stepping through the program, setting break points to analyse the code

ASP.Net tracing

Tracing is an activity to follow execution path and display the diagnostic information related to a specific Asp.Net web page or application that is being executed on the web server. Tracing can be enabled at development environment as well as in the production environment. These information can help you to investigate errors or unwanted results while ASP.NET processes a page request. You can view trace information at the bottom of individual pages and also you can use the trace viewer to view these trace information that is collected and cached by ASP.NET when tracing is enabled.

In Asp.Net Tracing is disabled by default. Trace statements are executed and shown only when tracing is enabled. You can enable tracing in two levels.

- Page Leve Tracing
- Application Level Tracing

You can enable individual pages as well as you can enabled your application's Web.config file to display trace information. When you enabled it application level, it displays all pages trace information unless the page explicitly disables tracing.

Page Leve Tracing

We can control whether tracing is enabled or disabled for an Asp.Net page with the Trace attribute of the @ Page directive.

It Shows all the general information about a web page when it is being processed. This is useful in debugging if a page does not work for any reason. Visual Studio provides detailed information about various aspects of the page and information such as the time for each method that is called in the web request.

For example, if your web application is having a performance issue, this information can help in debugging the problem. This information is displayed when the application runs in Visual Studio.

```
<%@ Page Trace="true" %>
```

```
<%@ Page Language="VB" Trace="true" AutoEventWireup="false" CodeFile="Default.aspx.vb" Inherits="_Default" %>
```

Application Level Tracing

When we enable application level tracing, trace information is gathered and processed for each page in that application. We can enable application level tracing by using the trace element in the Web.config file.

Application tracing allows one to see if any pages requested results in an error. When tracing is enabled, an extra page called trace.axd is added to the application. (See image below). This page is attached to the application. This page will show all the requests and their status.

```
<configuration>
  <system.web>
        <trace enabled="true" pageOutput="true" requestLimit="50"
        localOnly="false"  mostRecent="true" traceMode="SortByTime" />
  </system.web>
</configuration>
```

By default, application level tracing can be viewed only on the local Web server computer. The above configuration enables an application trace configuration that collects trace information for up to 50 requests.

Error Handling: Displaying a Custom Error Page

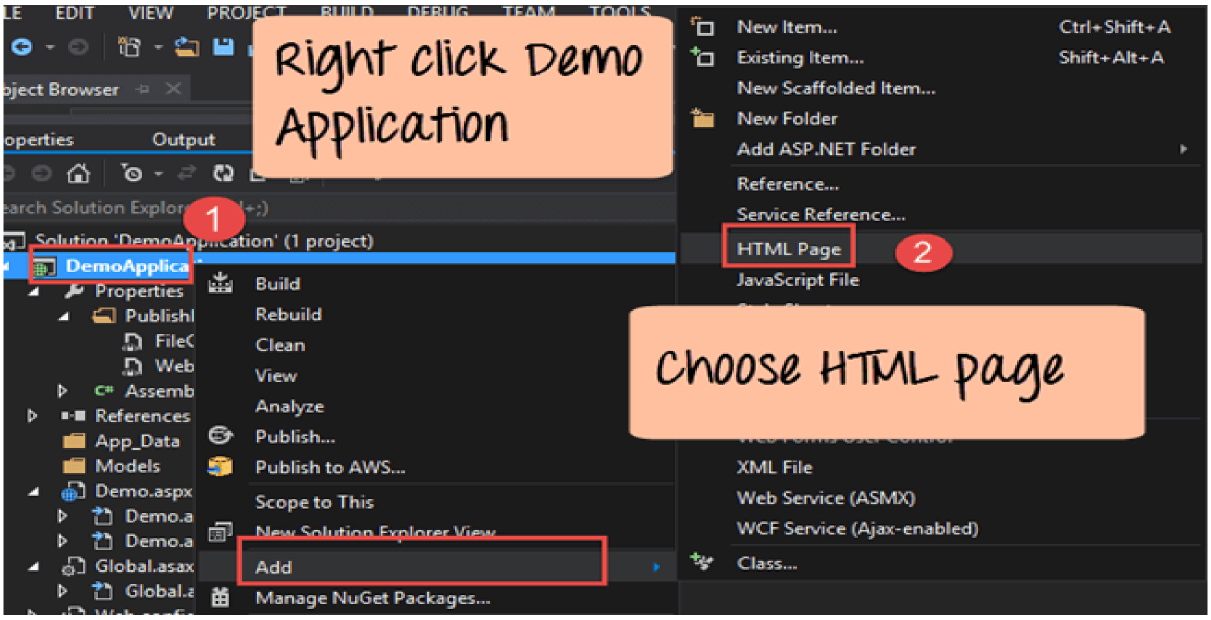
In ASP.NET, you can have custom error pages displayed to the users. If an application contains any sort of error, a custom page will display this error to the user.

In our example, we are first going to add an HTML page. This page will display a string to the user "We are looking into the problem". We will then add some error code to our demo.aspx page so that the error page is shown.

Let's follow the below mentioned steps

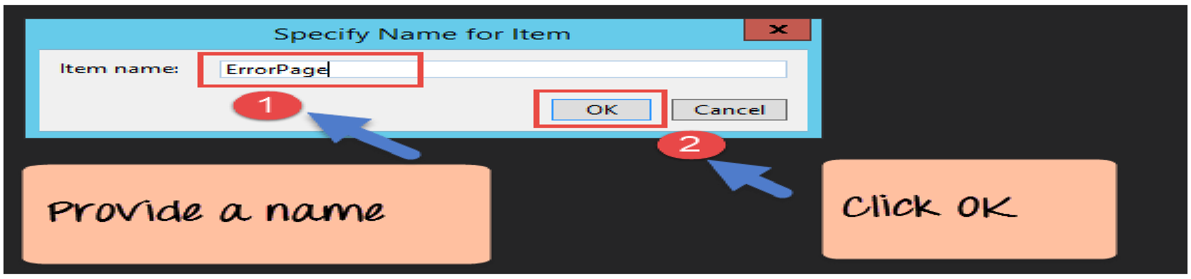
Step 1) Let's work on our DemoApplication. Let's add an HTML page to the application

1. Right-click on the DemoApplication in Solution Explorer
2. Choose the menu option 'Add'->HTML Page

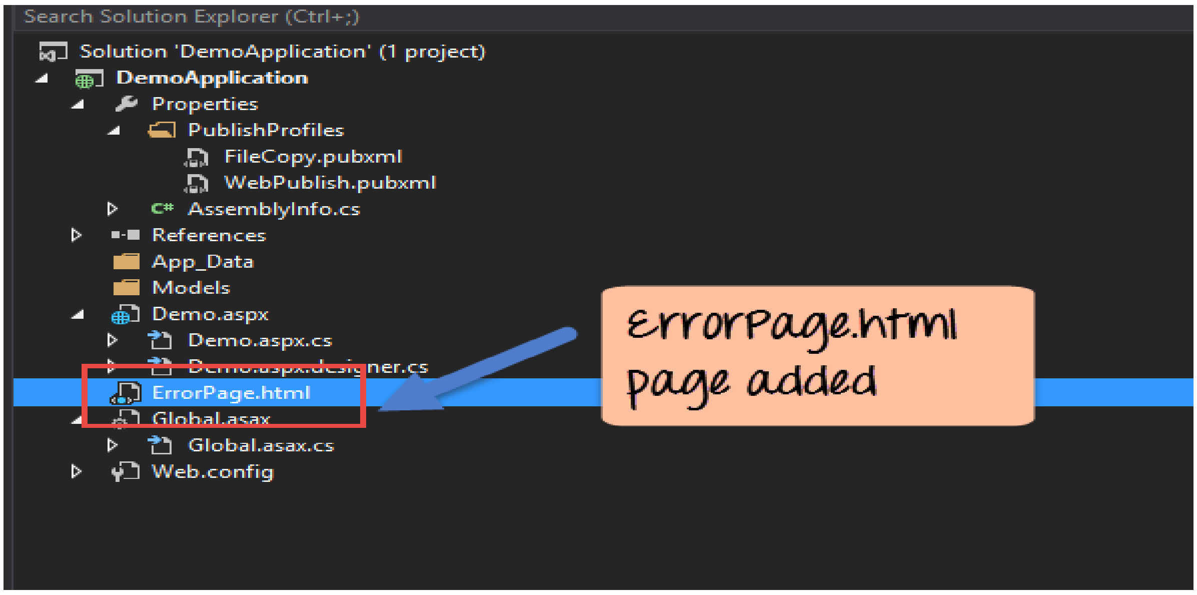


Step 2) In the next step, we need to provide a name to the new HTML page.

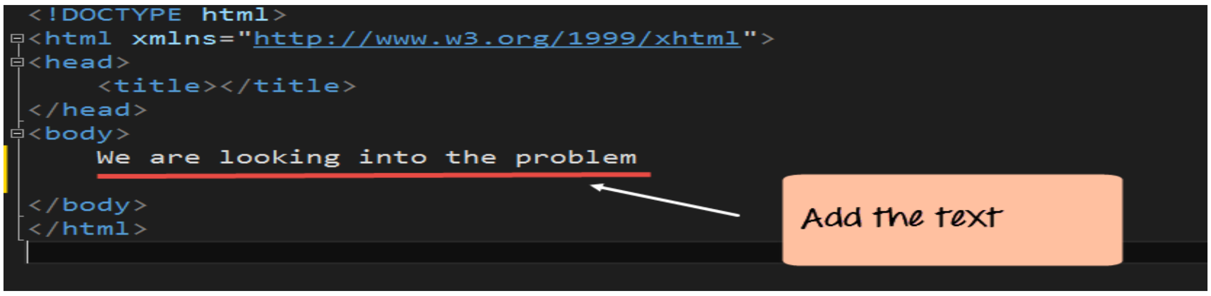
1. Provide the name as 'ErrorPage.'
2. Click the 'OK' button to proceed



Step 3) The Errorpage will automatically open in Visual Studio. If you go to the Solution Explorer, you will see the file added.



Add the code line “We are looking into the problem” to the HTML page. You don’t need to close the HTML file before making the change to the web.config file.

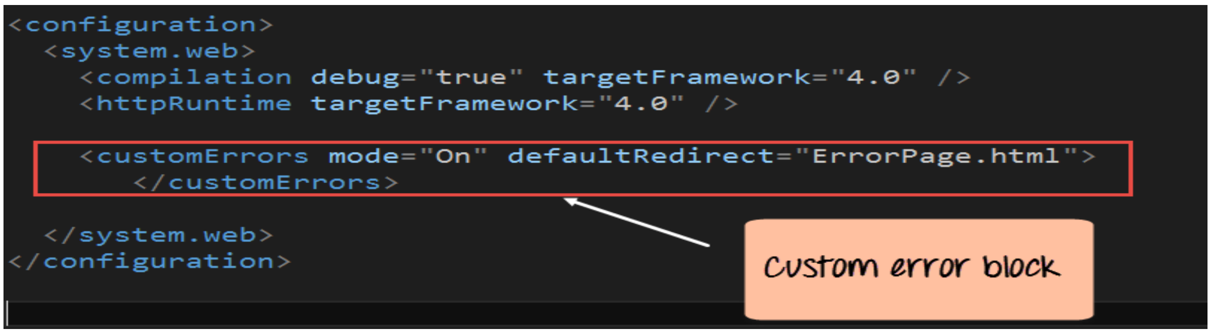


```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <title></title>
</head>
<body>
    We are looking into the problem

</body>
</html>
```

```
<!DOCTYPE html>
<html xmlns="http://www.w3.ore/1999/xhtml">
<head runat="server">
            <title></title>
</head>
            <body>
              We are looking into the problem
            </body>
</html>
```

Step 4) Now you need to make a change in the web.config file. This change will notify that whenever an error occurs in the application, the custom error page needs to be displayed.

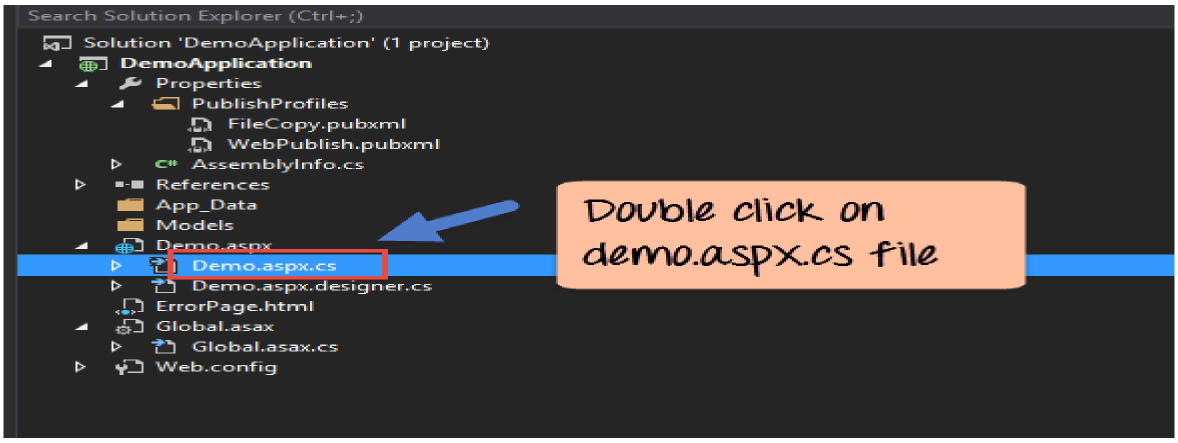
The ‘customErrors’ tag allows defining a custom error page. The defaultRedirect property is set to the name of our custom error’s page created in the previous step.



```
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0" />
    <httpRuntime targetFramework="4.0" />

    <customErrors mode="On" defaultRedirect="ErrorPage.html">
       </customErrors>

  </system.web>
</configuration>
```

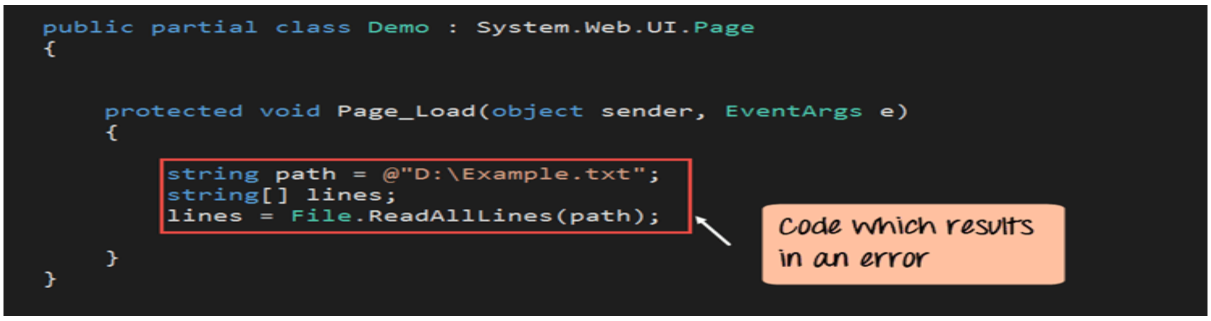
```
<configuration>
          <system.web>
                    <compilation debug="true" targetFramework="4.0" />
                    <httpRuntime targetFramework="4.0” />

                     <customErrors mode="On" defaultRedirect="ErrorPage.html">
</customErrors>


          </system.web>
</configuration>
```

Step 5) Now let’s add some faulty code to the demo.aspx.cs page. Open this page bydouble-clickingg the file in Solution Explorer



Add the below code to the Demo.aspx.cs file.

- These lines of code are designed to read the lines of a text from a file.
- The file is supposed to be located in the D drive with the name ‘Example.txt.’
- But in our situation, this file does not really exist. So this code will result in an error when the application runs.

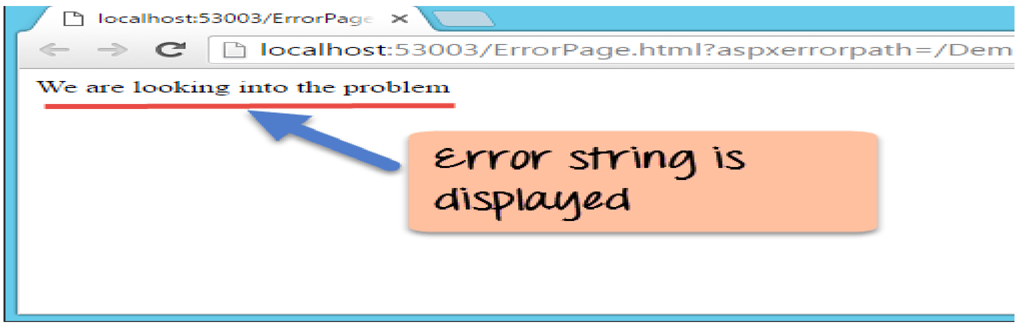


```
public partial class Demo : System.Web.UI.Page
{

    protected void Page_Load(object sender, EventArgs e)
    {
        string path = @"D:\Example.txt";
        string[] lines;
        lines = File.ReadAllLines(path);
    }
}
```

```
namespace DemoApplication
{

  public partial class Demo : System.Web.UI.Page
                          {
                            protected void Page_Load(object sender, EventArgs e)
                            {
                             String path = @"D:\Example.txt";
                             string[] lines;
                             lines = File.ReadAllLines(path);
                            }
                          }
}
```

Now execute the code in Visual Studio and you should get the below output.

Output:-



localhost:53003/ErrorPage
localhost:53003/ErrorPage.html?aspxerrorpath=/Dem
Error string is displayed

```
We are looking into the problem
```

The above page shows that an error was triggered in the application. As a result, the Error.html page is displayed to the user.

ASP.NET Unhandled Exception

Even in the best of scenarios, there can be cases of errors which are just not for seen.

Suppose if a user browses to the wrong page in the application. This is something that cannot be predicted. In such cases, ASP.Net can redirect the user to the errorpage.html.

ASP.NET Error logging

By logging application errors, it helps the developer to debug and resolve the error at a later point of time. ASP.Net has the facility to log errors. This is done in the Global.asax.cs file when the error is captured. During the capturing process, the error message can be written into a log file.

Summary

- ASP.Net has the facility to perform debugging and Error handling.
- Debugging can be achieved by adding breakpoints to the code. One then runs the Start with Debugging option in Visual Studio to debug the code.
- Tracing is the facility to provide more information while running the application. This can be done at the application or page level.
- At the page level, the code Trace=true needs to be added to the page directive.
- At the application level, an extra page called Trace.axd is created for the application. This provides all the necessary tracing information.