

ASP.NET 3

Med tveksam karma i C# 2.0 3.0

Hello world	5
PostBack och Page_Load med IsPostBack.....	10
Listkontroller	11
Klassvariabler ASP.NET	14
ViewState	16
Argumenten till händelsehanterare - object sender.....	17
Argumenten till händelsehanterare - EventArgs e.....	18
Filuppladdning.....	20
Server tags.....	23
MultiView	24
Valideringskontroller	26
Master Pages	30
Användarkontroller	35
Autentisering.....	42
DataBinding	46
Databaser	48
Repeater	49
Varför DataBinder.Eval behövs	51
Att använda data i en template.....	52
Avancerad <ItemTemplate>	54
UPDATE, DELETE, INSERT, etc.....	59
Att undvika SQL-injections	60
Att läsa från en databas utan databinding.....	61
Att centralisera anropen till databasen.....	63
Paging genom PagedDataSource	65
Paging med data från en databas.....	69
Avancerad Paging	71
ASP.NET AJAX	74
ASP.NET AJAX Control Toolkit	74
Aktivera ASP.NET AJAX med ScriptManager	75
UpdatePanel.....	75
UpdatePanel med Triggers.....	78
Att dela upp innehållet i flera UpdatePanels	78
ASP.NET AJAX från C#.....	80

UpdateProgress	84
Timer	85
<i>APPENDIX</i>	
Request och Response	90
Cookies	91
Sessioner	91

ASP.NET 3 - Med tveksam karma i C# ~~2.0~~ 3.0

ASP.NET är Microsofts ramverk för att utveckla webbapplikationer. Som namnet antyder är det en del av .NET-ramverket och har funnits tillgängligt sedan version 1.0 släpptes 2002. ASP.NET konkurrerar framförallt på företagsmarknaden med Java och bygger på en mer styrd utvecklingsplattform än exempelvis PHP. Detta har både för- och nackdelen att det är svårare för en utvecklare att sväva ut och göra egna lösningar på problem.

Webbsidor i ASP.NET kallas för *webforms* (även om den här texten för det mesta kallar dem ASP.NET-sidor) och är byggda runt ett system som mer liknar hur "vanliga" desktop-program utvecklas snarare än hur man traditionellt skriver hemsidor. En webform kan exempelvis "komma ihåg" hur den såg ut när data postades och automatiskt rendera om sig utan att utvecklaren behöver bry sig om det, precis som en utvecklare av ett vanligt desktop-program inte förväntar sig behöva rita om fönstret bara för att användaren klickat på en knapp.

Precis som desktop-program bygger ASP.NET på komponenter ("kontroller"), som exempelvis knappar, listor och textboxar. Många av de vanligaste komponenterna har direkta motsvarigheter i HTML, men några av de mer specialiserade kontrollerna, som exempelvis kalenderkontrollen, måste rendera stora mängder HTML-kod. Tidiga versioner av ASP.NET fick utstå mycket kritik för att kontrollerna renderade svårläst och dålig HTML, vilket dock har förbättrats avsevärt i de senare versionerna. Men faktum kvarstår att man som utvecklare är mer begränsad i hur HTML-koden i slutändan kommer se ut i en webform än man är i andra mer lättviktiga programspråk utan stora ramverk runt sig.

Som många andra Microsoft-lösningar är ASP.NET väldigt knutet till Windows-plattformen, och även om det teoretiskt går att skriva ASP.NET i andra editorer än Microsofts, och även om det går att använda ASP.NET med hjälp av tredje part i annan servermiljö än Windows, och även det går att ansluta till andra databaser än SQL-Server i ASP.NET, så finns det ytterst få företag och utvecklare som går den vägen. Kör man ASP.NET får man också räkna med att det i slutändan kommer vara Microsoft som gäller, och att det kommer kosta pengar. Eftersom ASP.NET framförallt har sin marknad kring kommersiell utveckling och företag är det också, tyvärr, naturligt att mycket av tredjepartsutvecklingen är kommersiell till skillnad mot PHP och andra språk med rötterna i öppen källkod.

ASP.NET är teoretiskt språkagnostiskt eftersom det bygger på .NET-ramverket, men så gott som all litteratur och exempelkod man stöter på använder sig antingen av C# eller VB.NET, och då Microsofts verktyg är så bundna till dessa två språken är det också ytterst få utvecklare som vågar sig på något annat.

Det här kompendiet strävar inte efter att vara en komplett guide till ASP.NET och är på det hela ganska dåligt. Du får kopiera det bäst du vill, men tänk på att papper kommer från att vi våldför oss på naturen, det finns alltid en PDF att tillgå också.

Hello world

Den här implementationen av *Hello world* i ASP.NET 2.0 består som de flesta ASP.NET-sidor av två separata filer:

hello.aspx – innehåller layout (HTML) och ASP.NET-kontroller som renderar sig till HTML.

hello.aspx.cs – innehåller C#-kod för att styra vad som händer på sidan.

När sidan körs från en webbläsare kompilerar ASP.NET-ramverket automatiskt de två filerna tillsammans. Ju mer komplex sidan blir desto mera tjänar man på att dela upp presentation och logik i två separata delar, men för ett så här triviale program kan det lätt uppfattas som tämligen bökigt,

```
<%@ Page Language="C#" CodeBehind="hello.aspx.cs" Inherits="Hello" %>
<html>
  <head>
    <title>Hello world i ASP.NET</title>
  </head>
  <body>
    <form runat="server">
      <div>
        <asp:label runat="server" id="Label1" Text="Svenska: Hej världen" />
        <br>
        <br>
        <asp:button runat="server" Text="Byt språk" OnClick="ChangeLanguage" />
      </div>
    </form>
  </body>
</html>
```

ASP.NET - hello.aspx

```
using System;
using System.Web.UI;

partial class Hello : Page
{
    protected void ChangeLanguage(object sender, EventArgs e)
    {
        Label1.Text = "English: Hello world";
    }
}
```

C# - hello.aspx.cs

Alternativt kan allt skrivas på en sida, klassdeklarationen försvinner då helt. Det är relativt sällan man ser den här varianten och är inget vi kommer använda i det här kompendiet,

```
<%@ Page Language="C#" %>
<%@ Import Namespace="System" %>
<%@ Import Namespace="System.Web" %>

<script runat="server">
protected void ChangeLanguage(object sender, EventArgs e)
{
    Label1.Text = "English: Hello world";
}
</script>

<html>
  <head><title>Hello world i ASP.NET</title></head>
  <body>
    <form runat="server">
      ...
    </form>
  </body>
</html>
```

ASP.NET och C# - hello2.aspx

hello.aspx

```
<%@Page
```

Page directive för en ASP.NET-sida, sidans konfiguration ungefär.

```
CodeBehind="hello.aspx.cs"
```

Vilken fil som skall kompileras tillsammans med sidan.

```
Inherits="Hello"%>
```

Vad *klassen* heter som den här sidan skall *ärva* från (= använda sig av).

```
<form runat="server">
```

Om sidan använder sig av ASP.NETs *webcontrols* måste dessa av tekniska skäl ligga inuti en `<form>` som är `runat="server"`. Det kan bara finnas en `<form>` av denna typ per sida och den skrivs därför vanligen direkt efter `<body>`.

```
<asp:label runat="server"
```

En ASP.NET *webcontrol* av typen `Label`, används vanligen för att skriva ut text och renderas i webbläsaren som innehållet en `span`-tag.

```
id="Label1"
```

Ett egendefinierat ID som används för att "komma åt" kontrollen från C#-koden.

```
Text="Svenska: Hej Världen"/>
```

Attributet `Text` är en *egenskap* på kontrollen för vilken text som skall skrivas ut i `span`-taggen när den renderas.

```
<asp:button runat="server
```

En kontroll för att rendera en knapp som "postar tillbaka" sidan till servern. Då vi inte någonstans i C#-koden kommer behöva "komma åt" knappen kan vi utesluta ID-attributet.

```
Text="Byt språk"
```

Egenskapen `Text` på en `button`-kontroll anger vilken text som står skrivet på den.

```
OnClick="ChangeLanguage"/>
```

`onClick` är ett attribut som markerar en händelse med `ChangeLanguage` definierad som namnet på den *händelsehanterare* som skall aktiveras. Det kan utläsas som "När knappen klickas kör metoden `ChangeLanguage`". Metoden `ChangeLanguage` måste nu vara definierad i C# annars misslyckas kompileringen av sidan.

using System.Web.UI

Använder sig av ASP.NET-klasserna i System.Web.UI

partial

partial markerar att klassen är uppdelad i flera delar. Den andra delen skapas automatiskt av ASP.NET och innehåller bland annat genererade instansvariabler för varje kontroll på sidan med ett ID (i det här fallet finns bara en, Label1).

class Hello : Page

Sidans klass heter Hello och *ärver* från Page som kommer från System.Web.UI. Alla ASP.NET-sidor måste ära från den här klassen för att systemet skall fungera. Klassnamnet måste överensstämma med vad som står i Inherits högst upp i *Page directive*.

protected void

Alla metoder som svarar på en händelse, som i det här fallet när användaren klickar på en knapp, returnerar **void** och är, som standard, deklarerade med **protected** som modifierare (rent tekniskt fungerar alla modifierare här utom **private**)

ChangeLanguage

Namnet på metoden måste överensstämma med vad som står i kontrollerna på ASPX-sidan...

(**object** sender, EventArgs e)

...och ha rätt typ av argument. Alla händelsehanterare tar en viss typ av argument och detta är den absolut simplaste varianten, vilket alltså onclick använder sig av. Objektet sender är alltid den kontroll som skapat händelsen, vilket kan vara användbart om flera olika kontroller använder samma händelsehanterare. Det andra argumentet kan variera i typ och är eventuell extra information som skickats med händelsen. I det här fallet är den av typen EventArgs som innebär att ingen extra information har skickats med.

En mer detaljerad beskrivning finns på sidorna 18 och 19.

Label1.Text = "English ...

Ändrar Text-egenskapen på den automatiskt instansierade variabeln Label1 som skapats i bakgrunden och överensstämmer med det ID som kontrollen har på ASPX-sidan.

ASP.NET Uppgift 1-1

Starta ett nytt webbprojekt och skriv *Hello World* som beskrivet på tidigare sidor, men ändra så att man kan växla språk fram och tillbaka genom att klicka på knappen flera gånger. För detta kan du lämpligen jämföra texten på Label-kontrollen (om den är på svenska, ändra till engelska, och vice versa).

Använd **F5** för att kompilera och provköra sidan. Du kan göra vissa ändringar i ASPX-filen utan att behöva kompilera om sidan, men för att ändra i C#-koden måste du stoppa debug-processen (stop-knappen i toolbar-menyn eller **shift+F5**). Efter du har stoppat debug-processen kan du göra ändringar och sedan kompilera om sidan med **ctrl+shift+F5** och ladda om den i webbläsaren. Efter du har stängt ner debug-processen fungerar inte dina eventuella *breakpoints*, och *exceptions* kommer inte fångas upp i editorn förrän du kompilerar om med en debugger på nytt (**F5**).

Försök lära dig ovanstående process då du kommer nyttja den i så gott som samtliga uppgifter framöver.

ASP.NET Uppgift 1-2

Försäkra dig om att projektet inte är i debug-läge och lägg till en ny sida genom att högerklicka på projektnamnet i *Solution Explorer* och välj *Add -> New Item... -> Web Form*. Kalla sidan för uppgift1-2.aspx eller liknande fantasifullt. Gör sidan till startsida (det vill säga den sidan som kommer upp när du kompilerar projektet med **F5**) genom att högerklicka på namnet och välja *Set As Start Page*.

Skapa en sida med två textboxar. En textbox-kontroll renderas till en input-tag i HTML och ser i ASP.NET ut något såhär ungefär,

```
<asp:textbox id="text1" runat="server" text="Strawberry fields forever..." />
```

Skapa därefter en knapp med texten "Jämför" och en tom Label-kontroll. När användaren klickar på knappen skall texterna i textboxarna jämföras och ett av följande meddelande skrivs till Label-kontrollen.

- Duktigt, texterna är identiska.
- Fel! Texterna är INTE identiska.
- Du har glömt skriva något i textboxarna.

Och vips har du skapat en spännande leksak lämplig för aporna på kolmårdens djurpark.

Skriv samma sak i båda textboxarna

Text 1

Text 2

Fel! Texterna är INTE identiska.

ASP.NET Uppgift 1-3

Uppdatera Uppgift 1-2 så att texten som skrivs ut blir röd om textboxarna inte är identiska och grön om den är det. Det finns ett flertal sätt att göra detta på, men lättast är att använda sig av egenskapen `ForeColor` på `Label`-kontrollen.

Tyvärr är det inte så enkelt att man bara kan skriva in en sträng med rätt hexkod för den färgen man vill använda, utan man tvingas istället gå via `System.Drawing.Color`. Lyckligtvis finns det en "översättare" i `System.Drawing.ColorTranslator` som kan konvertera om hex till rätt färg i ASP.NET, vilket naturligtvis är att föredra om man inte vill begränsa sig till abstrakta och larviga färgnamn som "Honeydew" och "MediumSeaGreen". Följande två rader gör samma sak,

```
Label1.ForeColor = System.Drawing.Color.white; // vit text
Label1.ForeColor = System.Drawing.ColorTranslator.FromHtml("#FFFFFF"); // vit text
```

ASP.NET Uppgift 1-4

Det är fullt möjligt att applicera `runat="server"` på vanliga HTML-element. Dessa blir då så kallade *HtmlControls*, vilka skiljer sig från ASP.NETs kontroller (*WebControls*) med att de inte kan ha händelser och har en lägre grad av abstraktion (en `<h1 runat="server">` kommer alltid rendera sig som `h1`-tag medan en *WebControl* kan rendera hela klumpar med HTML och teoretiskt anpassa sig efter vilken kontext den befinner sig i).

En *HtmlControl* delar en hel del egenskaper med en *WebControl* då de båda ärver från samma basklasser. För utvecklaren är en *HtmlControl* ett smidigt sätt att komma åt och manipulera de vanliga HTML-taggar från C#-koden.

Skapa en ny sida som innehåller två textboxar där användaren kan skriva in sidans titel och sidans bakgrundsfärg, samt en knapp som genomför själva bytandet. En *HtmlControl* är som sagt mer lättviktig när det gäller sin abstraktion och använder sig inte av `System.Drawing.Color` för sina färger.

För att ändra bakgrundsfärgen sätter du ett `id` och ett `runat="server"` på `<body>` som du sedan kan komma åt i metoden för `onclick`-händelsen på knappen. För att faktiskt ändra ett attribut kan du antingen gå genom den indexerade arrayen `Attributes`,

```
bodytaggen.Attributes["bgcolor"] = "#ff0000"; // HTML 3.2! Oldschool! uh...
```

...eller använda dig av `style`-attributet som fungerar på samma sätt. Med hjälp av Visual Webdevelopers utmärkta *Intellisense* kan du gå igenom listan med egenskaper och metoder för objektet och leta upp vad som finns att tillgå.

För att ändra sidans `<title>` räcker det med att du har kvar `runat="server"` på `<head>` för att ASP.NET automatiskt kommer generera ett `Header`-objekt (du behöver alltså inte själv ange `id="Header"` i `<head>`) med vilket du kan ändra sidans titel. Använd *Intellisense* för att klura ut exakt hur, att lära sig använda editorn och dess funktioner effektivt är A och O för ett så komplicerat ramverk som ASP.NET faktiskt är.

PostBack och Page_Load med IsPostBack

Ett viktigt begrepp i ASP.NET är *PostBack*, en sekvens där sidan postar sitt formulär till sig själv. Vid en PostBack renderar ASP.NET om sidan och eventuella händelser som har aktiverats av användaren kopplas ihop med rätt metoder. Det är vanligt att bara en händelse (som en knapptryckning) behöver hanteras, men teoretiskt kan flera olika fördröjda händelser som inte aktiverar en PostBack av sig själva också hanteras, som exempelvis en dropdown-lista vars värde kan ha ändrats sedan förra gången sidan postades.

En ASP.NET-sida har flertalet olika "magiska" metoder som, om de finns definierade, automatiskt anropas när en sida renderas. Den mest använda metoden är *Page_Load* som anropas *före* eventuella andra händelsehanterare har sparkat igång. Den är till och med så vanlig att Visual Webdeveloper automatiskt skriver dit en tom *Page_Load* i C#-koden när man skapar en ny sida.

Ett vanligt scenario är att man vill göra något när sidan första gången laddas in, men inte påföljande gånger som sidan postar tillbaka till sig själv. Ett typiskt exempel skulle kunna vara att man första gången sidan renderas vill fylla i fält med någon standardtext, men när sidan postas tillbaka vill man inte skriva in texten igen om användaren skulle ha ändrat den. För att lösa detta har varje sida en boolesk egenskap i *IsPostBack* som helt enkelt talar om huruvida sidan postats tillbaka (**true**) eller om det här är första gången sidan renderas (**false**).

```
<form runat="server">
  <pre>
    Time: <asp:label runat="server" id="Label1" Text="" />
    Postbacks: <asp:label runat="server" id="Label2" Text="" />
    <asp:button runat="server" Text="PostBack!" onClick="DoPostBack" />
  </pre>
</form>
```

ASP.NET

```
partial class PostbackTest : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if ( ! IsPostBack)
        {
            Label1.Text = DateTime.Now.ToString();
            Label2.Text = "0";
        }
    }

    protected void DoPostBack(object sender, EventArgs e)
    {
        int i = Convert.ToInt32(Label2.Text);
        i++;

        Label2.Text = Convert.ToString(i);
    }
}
```

```
Time: 2009-04-24 22:09:28
Postbacks: 0
PostBack!
```

```
Time: 2009-04-24 22:09:28
Postbacks: 7
PostBack!
```

C# - Oavsett hur många gånger sidan postar tillbaka till sig själv kommer `Label1.Text` alltid bara ha tiden då sidan genererades första gången.

Listkontroller

Listor i ASP.NET skiljer sig lite från hur vi vanligen använder dem i HTML. Till skillnad mot HTML har ASP.NET en gemensam grund som alla listor bygger på, oavsett om det är dropdown-listor, listboxar, eller listor med radioknappar eller checkboxar (något som inte ens finns som enskilda element i HTML). Alla listor i ASP.NET bygger på nedanstående struktur,

```
<asp:typ-av-lista runat="server" id="lista">
  <asp:ListItem>rad 1</asp:ListItem>
  <asp:ListItem>rad 2</asp:ListItem>
  <asp:ListItem>rad 3</asp:ListItem>
</asp:typ-av-lista>
```

ASP.NET

Typ-av-lista i exemplet ovan kan exempelvis vara `RadioButtonList`, `CheckBoxList`, `ListBox`, `DropDownList`, med mera, och renderas därefter; en `DropDownList` blir en `<select>` och en `RadioButtonList` en simpel `<table>` med radioknappar på varje rad, osv. Eftersom alla listorna bygger på samma grundstruktur fungerar de också ungefär likadant och är från en utvecklares synpunkt relativt enkla att använda. De är inte helt identiska dock, exempelvis har en `ListBox` attributet `rows` som talar om hur många rader som skall visas, något som - av naturliga skäl - inte återfinns i exempelvis en `DropDownList` som alltid bara visar en rad.

Alla listor med innehåll har en eller flera `ListItem`, och det kan vara värt att notera att dessa element alltså *inte* skall ha ett `runat="server"` då de sägs tillhöra listorna (som i sin tur har attributet). Ett `ListItem` fungerar likadant oavsett vilken typ av lista som används och representerar som torde vara uppenbart en enskild rad i listan. För att komma åt och manipulera raderna från C# används, på listan, egenskapen `Items` som innehåller en kollektion (se sidan 43 i *Rädsla och Avsky*) med samtliga `ListItems` som den består av,

```
protected void Page_Load(object sender, EventArgs e)
{
    lista.Items[1].Text = "Karma Police"; // Ändrar texten på andra raden i listan
    lista.Items.Remove(lista.Items[0]); // Tar bort första raden i listan
    int rows = lista.Items.Count;      // Sätter rows till 2
}
```

C#

Ett intressant attribut som går att sätta på listor är `AutoPostBack="true"`, vilket genererar en liten snutt JavaScript som ser till att sidan automatiskt gör en `PostBack` när något i listan ändrar sig (som att man klickar i en checkbox i en lista med checkboxar eller väljer ett nytt alternativ i en dropdown-lista). För att faktiskt fånga upp vilket val användaren har gjort i listan kan man binda en händelsehanterare till händelsen `onSelectedIndexChanged` (se exemplet på nästa sida).

Nedanstående kod implementerar en DropDownList med djur. När användaren väljer ett djur gör ASP.NET en PostBack och SelectNewAnimal-metoden anropas. Ovanför listan finns en Label där "valt djur" skrivs ut, bredvid en knapp där man kan ta bort aktuellt djur från listan, och under en textbox med en knapp som lägger till ett nytt djur i listan och väljer det.

```
<form runat="server">
  Valt djur: <asp:Label runat="server" ID="SelectedAnimal" />
  <hr />
  <table>
  <tr>
  <td>
    <asp:DropDownList runat="server" width="100" ID="Animals"
      AutoPostBack="true" OnSelectedIndexChanged="SelectNewAnimal">
      <asp:ListItem>Skalbagge</asp:ListItem>
      <asp:ListItem>Zombiekatt</asp:ListItem>
      <asp:ListItem>Jagular</asp:ListItem>
    </asp:DropDownList>
  </td>
  <td><asp:Button runat="server" Text="Ta bort" OnClick="DeleteAnimal" /></td>
  </tr>
  <tr>
  <td><asp:TextBox ID="AnimalToAdd" runat="server" width="100"/></td>
  <td><asp:Button runat="server" Text="Lägg till" OnClick="AddNewAnimal"/></td>
  </tr>
  </table>
</form>
```

ASP.NET

```
public partial class Listor : System.Web.UI.Page
{
  protected void Page_Load(object sender, EventArgs e)
  {
    if (!IsPostBack)
      UpdateSelectedAnimal();
  }

  protected void DeleteAnimal(object sender, EventArgs e)
  {
    Animals.Items.Remove(Animals.SelectedItem);
    UpdateSelectedAnimal();
  }

  // Lägger till ett nytt djur sist i listan och väljer det
  protected void AddNewAnimal(object sender, EventArgs e)
  {
    Animals.Items.Add(AnimalToAdd.Text);
    Animals.SelectedIndex = Animals.Items.Count - 1;
    AnimalToAdd.Text = "";
    UpdateSelectedAnimal();
  }

  protected void SelectNewAnimal(object sender, EventArgs e)
  {
    UpdateSelectedAnimal();
  }

  // Uppdaterar "valt djur:"-texten
  private void UpdateSelectedAnimal()
  {
    // Har vi tagit bort alla djur är SelectedItem null och programmet
    // kraschar om vi försöker använda .Text på ett null-objekt
    if (Animals.Items.Count == 0)
      SelectedAnimal.Text = "";
    else
      SelectedAnimal.Text = Animals.SelectedItem.Text;
  }
}
```

C#

The screenshot shows a web form with the following elements:

- A label "Valt djur: Elefant" at the top.
- A dropdown menu below the label, currently displaying "Elefant".
- A button labeled "Ta bort" to the right of the dropdown.
- A text input field below the dropdown.
- A button labeled "Lägg till" to the right of the text input field.

ASP.NET Uppgift 2-1

Surfa in på Wikipedia och leta upp en lagom lång artikel. Klistra in och formatera texten så att den inte ser för tokig ut, du kan välja att korta ner innehållet om det är överflödigt mycket eller svårformaterat.

Lägg in allt innehåll inklusive rubriker i olika `<asp:Label>` (`<asp:placeholder>` fungerar också och är mer korrekt om du inte vill generera en `` runt innehållet). Sätt `visible="false"` så att man bara ser ingressen och titeln på sidan när man först surfar in på den.

Skapa en `checkboxlist` med ett `Listitem` för varje sektion med innehåll. Sätt `autopostback="true"` och skriv en händelsehanterare för `onselectedindexchanged` som döljer/visar innehållet (sätter `visible` till `true`) beroende på vilka checkboxar som är valda, enligt nedanstående illustration,

Tetris



Tetris (ryska:Тетрис) är ett dator- och TV-spel som går ut på att ordna olika fallande figurer. Tetris uppfanns 1985 av den ryske matematikern Aleksej Pazjtnov när han arbetade på Vetenskapsakademien i Moskva.

- 1 Spelet
- 2 Gravitation
- 3 Poängräkning
- 4 Spelstrategier
- 5 Musik

Spelet
'Tetris' syftar på att alla block i spelet är uppbyggda av fyra rutor. Det finns sju möjliga, sammanhängande figurer som består av fyra rutor vardera. Dessa kallas ofta för "I", "T", "O", "L", "J", "S" och "Z".

Poängräkning
För att belöna "svårare" radborttagningar får man mer poäng om man tar bort flera rader med ett och samma block.

Det finns en hel del olika sätt att lösa den här uppgiften på men troligen måste du gå igenom din checkbox-lista med exempelvis en for-loop och kontrollera varje enskilt `Listitem` ett i taget huruvida de är ibockade eller inte (egenskapen `selected`). På listan finns visserligen också egenskapen `selectedIndex`, men eftersom fler än ett `Listitem` kan vara valt åt gången kommer `selectedIndex` alltid bara peka den sista valda checkboxen i listan.

Klassvariabler ASP.NET

En klassvariabel i ASP.NET, alltså en variabel markerad med **static** (sidan 32 i *Rädsla och Avsky*) utanför någon metod, fungerar annorlunda jämfört med klassvariabler i exempelvis PHP. En klassvariabel i ASP.NET har den egenskapen att den inte är unik för varje förfrågan, så som vi är vana vid, utan den lever kvar i klassen tills ASP.NET-processen startas om (vilket görs automatiskt lite då och då och naturligtvis när webbservern startas om). När ASP.NET kompilerar ett projekt så skapas en DLL-fil som används av webbservern, klasserna i denna DLL lever precis som vore de kompilerade för ett "vanligt" program; det vill säga de kompileras inte om för varje sidförfrågning såvida inte utvecklaren har gjort en förändring. Resultatet blir att en klassvariabel behåller sitt värde så länge som processen lever kvar, eller med enklare ord - alla användare "delar" på variabeln.

Med denna insikt står det förhoppningsvis klart att man skall vara ytterst försiktig med att använda klassvariabler i ASP.NET, och helst undvika dem helt om man är osäker. Skulle flera användare vara inne på sidan och uppdatera variabeln samtidigt kan ett *race condition* uppstå om man inte är försiktig. I exemplet nedan har vi en **static int** *x* som användaren kan sätta via en textbox. Låt säga att användaren tillderar *x* värdet 10 som sedan skrivs tillbaka till sidan i en label. Om en annan användare är inne på sidan samtidigt och i sin tur hinner uppdatera *x* med ett annat värde innan värdet som den första användaren skrev (10) har hunnit skrivas tillbaka till sidan resulterar det i att den första användaren får det nya värdet istället för det han eller hon skrev in.

```
<form runat="server">
  <asp:TextBox runat="server" ID="UserInput" />
  <asp:button runat="server" onClick="UpdateX" text="Update!" />
  <br />
  Värdet av X = <asp:Label runat="server" ID="ValueOfX" />
</form>
```

ASP.NET

```
public partial class RaceTest : System.Web.UI.Page
{
    protected static int x;

    protected void UpdateX(object sender, EventArgs e)
    {
        x = Convert.ToInt32(UserInput.Text);

        // *** Race condition ***
        System.Threading.Thread.Sleep(10000);

        valueOfX.Text = x.ToString();
    }
}
```

C# - För att öka möjligheterna att ett race condition inträffar används metoden Thread.Sleep för att stanna upp exekveringen i tio sekunder i det kritiska läget, på den tiden hinner vi öppna en ny webbläsare och prova att skriva in ett annat värde i textboxen.

Klassvariabler är dock inte helt av ondo om de bara används på rätt sätt, de kan exempelvis hålla ett globalt objekt som alla användare kan dela på (som ett Random-objekt för att generera slumptal). Är man det minsta osäker är det dock säkrast att undvika **static** över huvudtaget eftersom man kan skjuta sig själv i foten ganska rejält med egendomliga fel som uppstår först när många användare använder sidan samtidigt.

ASP.NET Uppgift 2-2

Skriv en *tjötbox*, en väldigt simpel och korkad variant av en webbchat.

Använd en klassvariabel (**static**) för att spara konversationerna; när en person klickar på "uppdatera" lägger du helt enkelt till eventuellt nytt tjöt till slutet av strängen tillsammans med en radmatning, tidsangivelse och namnet på den som tjöta.

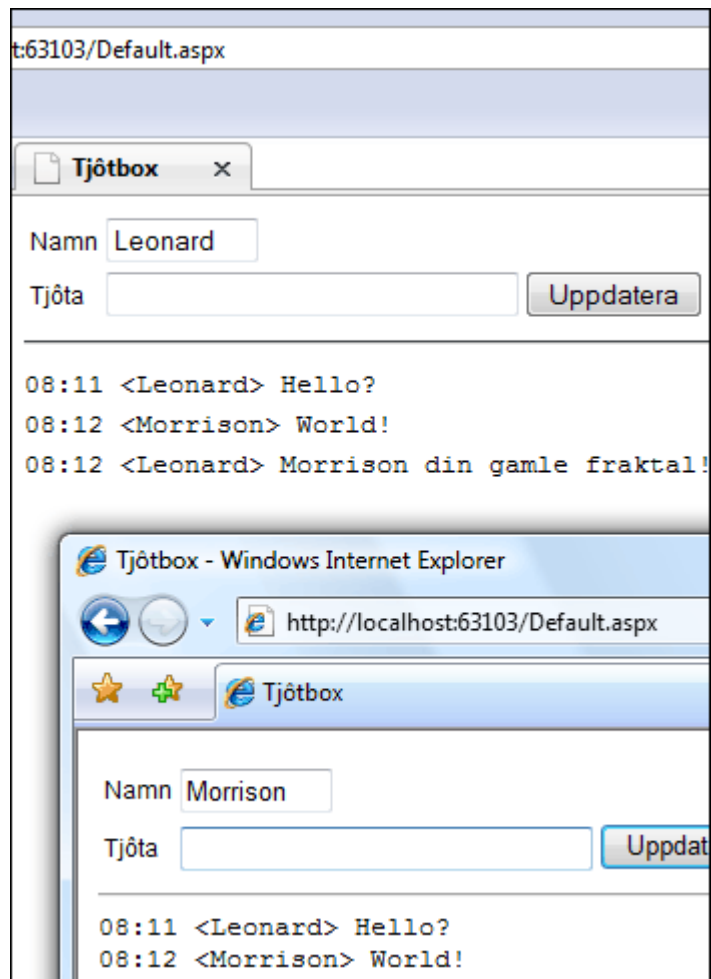
När en användare först surfar in på sidan skall eventuell tidigare konversation skrivas ut.

Eftersom en statisk variabel inte är unik kommer alla användare se varandras tjöt så länge sidan inte kompileras om eller något annat gör att ASP.NET-processen måste startas om.

Använd en `Label` för att skriva ut strängen till användaren.

För att få fram tidsangivelse i timmar och minuter, så som i illustrationen, kan man använda nedanstående:

```
string timestamp =  
    DateTime.Now.Hour.ToString().PadLeft(2, '0') + ":" +  
    DateTime.Now.Minute.ToString().PadLeft(2, '0')
```



ASP.NET Uppgift 2-3 (frivillig)

Utveckla tjötboxen vidare, lämpligen se till så att konversationen inte kan bli längre än ett visst antal rader. Till detta kan du exempelvis istället för en enda stor sträng använda en statisk array med strängar som när arrayen är "full" tar bort det översta elementet och flyttar alla andra ett steg uppåt. Detta kan kräva en hel del klurande dock.

ViewState

I ASP.NET används konceptet *ViewState* för att lagra information mellan PostBacks på en sida. Eftersom HTTP i sig inte har någon mekanism för att "komma ihåg" hur en sida ser ut mellan olika förfrågningar använder sig ASP.NET av en lång sträng i ett dolt input-fält för detta ändamålet. Denna sträng, ViewState-strängen, innehåller bland annat information om de delarna på sidan som inte automatiskt kommer med när man postar ett formulär. Funktionen för ViewState behövs exempelvis inte för att ASP.NET skall kunna komma ihåg vad som står i en textbox (formulärdata postas ju alltid automatiskt i HTML), däremot behövs den för att ASP.NET skall kunna komma ihåg vad som stod i en `<asp:label>`, eller ``-tag som den blir renderad som.

ViewState-strängen kan lätt bli väldigt lång, i extrema fall upp till flera hundra kilobyte om man inte är försiktig. Då ViewState-strängen vanligen renderas högt upp på sidan kan det av användaren lätt uppfattas som att sidan blir långsam när ViewState har vuxit sig stor. Av denna anledning är det möjligt att stänga av ViewState med hjälp av attributet `EnableViewState="false"`, antingen på enstaka kontroller eller på hela sidan genom att skriva det i siddirektivet högst upp.

ViewState sker helt automatiskt för utvecklaren och ASP.NET sköter vanligen renderingen av kontroller vid PostBack utan att någon som helst handpåläggning behövs. Det är också fullt möjligt att själv utnyttja ViewState-strängen till att spara ner egna värden, som då fungerar ungefär som en typ av sessionsvariabler, med skillnaden att de bara existerar på den aktuella sida (ViewState är unik för varje sida och det är inte möjligt att posta med ViewState-data från en sida till en annan).

I ViewState kan man spara ner alla typer av objekt som går att *serialisera*, exempelvis strängar, heltal, arrayer, mer mera. Då ViewState inte kan hålla reda på vilken typ av data som man sparar ner är det upp till en själv som utvecklare att konvertera tillbaka till rätt datatyp med en *typecast* (sidan 15-16 i *Rädsla och Avsky*) när man sedan plockar ut objektet igen,

```
<form runat="server">
  Another Visitor...<br />
  Klicka <asp:linkbutton runat="server" Text="här" onClick="DoPostBack" />
  för att se hur mycket tid du slösat bort!<br/>
  <asp:label runat="server" id="Label1" Text="" Font-Bold="true"/>
</form>
```

ASP.NET

```
partial class ViewStateTest : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if ( ! IsPostBack) // Sparar ner ett DateTime-objekt med aktuell tid
            ViewState["timestamp"] = DateTime.Now;
    }

    protected void DoPostBack(object sender, EventArgs e)
    {
        DateTime now = DateTime.Now;
        DateTime then = (DateTime)ViewState["timestamp"]; // och hämtar ut det igen
        TimeSpan span = now.Subtract(then);

        Label1.Text = "Du har varit på den här sidan i "
            + span.TotalSeconds + " sekunder.";
    }
}
```

C#

```
Another Visitor...
Klicka här för att se hur mycket tid du slösat bort!
Du har varit på den här sidan i 4,241 sekunder.
```

Argumenten till händelsehanterare - object sender

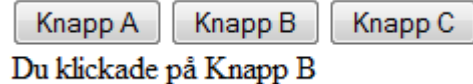
Våra händelsehanterare för `onClick` har än så länge alla följt samma mönster; varje metod har tagit ett anonymt objekt som första argument och ett objekt av typen `EventArgs` som det andra. Det första argumentet är alltid ett objekt av den kontroll därifrån händelsen härstammar, det vill säga när man klickar på exempelvis en knapp så är första objektet i händelsehanteraren en instans av den knappen. Då samma händelsehanterare kan användas av flera olika kontroller är det inte möjligt att i förväg veta vilken datatyp som första argumentet är av, utan vi tvingas manuellt typecasta till rätt typ om vi vill ha ut kontrollen i C#-koden,

```
<form runat="server">
  <asp:Button runat="server" OnClick="ButtonClick" Text="Knapp A"/>
  <asp:Button runat="server" OnClick="ButtonClick" Text="Knapp B"/>
  <asp:Button runat="server" OnClick="ButtonClick" Text="Knapp C"/>
  <br />
  <asp:Label runat="server" ID="whatButton" />
</form>
```

ASP.NET - Alla tre knappar använder samma händelsehanterare.

```
public partial class ArgumentTest : System.Web.UI.Page
{
    protected void ButtonClick(object sender, EventArgs e)
    {
        // Typecasta sender till en Button-kontroll
        Button b = (Button)sender;
        whatButton.Text = "Du klickade på " + b.Text;
    }
}
```

C#



Ovanstående kod fungerar bara så länge som det verkligen *är* en `Button` som har aktiverat händelsehanteraren. Skulle vi exempelvis byta ut en av knapparna till en `LinkButton` skulle programmet krascha med en `InvalidCastException` när vi klickade på den. Vi får alltså ingen hjälp av kompilatorn här utan får själva hålla reda på vilken datatyp som gäller. Det är visserligen ganska ovanligt men det förekommer tillfällen då man behöver testa om `sender` är av en viss datatyp innan man försöker sig på en typecast, för att försäkra sig om att det inte kommer krascha programmet,

```
protected void ButtonClick(object sender, EventArgs e)
{
    if (sender is Button)
    {
        Button b = (Button)sender;
        whatButton.Text = "Du klickade på en Button, " + b.Text;
    }
    else if (sender is LinkButton)
    {
        LinkButton b = (LinkButton)sender;
        whatButton.Text = "Du klickade på en LinkButton, " + b.Text;
    }
    else
    {
        whatButton.Text = "Du klickade på något annat...";
    }
}
```

C# - Med hjälp av det reserverade ordet `is` kan vi testa om ett objekt är av en viss datatyp.

Argumenten till händelsehanterare - EventArgs e

Det andra argumentet till händelsehanteraren, EventArgs, är helt meningslöst och har än så länge inte innehållit någon information alls då onclick är en så väldigt enkel händelse. Anledningen varför EventArgs måste vara med överhuvudtaget är rent teknisk och bygger på att andra, mer avancerade händelser, skall kunna bygga vidare (ärva av) EventArgs och i sin tur kunna skicka med extra information som gäller den aktuella händelsen.

En sådan händelse är onCommand, som på en knapp är ett alternativt sätt att fånga upp när en användare klickar på den. Varför det finns två olika sätt att göra samma sak beror på att för det mesta så vill utvecklaren bara ha reda på om någon klickade på knappen, och i de fallen duger den enklare onclick bra. I ett mer avancerat exempel skulle det exempelvis kunna finnas en "Köp"-knapp bredvid en lista med varor i en webbshop. Alla "Köp"-knappar använder samma händelsehanterare men utvecklaren måste ha med vilket ProduktID som knappen representerar för att kunna hämta ut rätt vara från databasen och lägga till den i varukorgen.

En oerfaren utvecklare skulle kanske tycka att det vore lättare att bara skapa en händelsehanterare för varje produkt som går att köpa, men det fungerar naturligtvis inte i någon större utsträckning och definitivt inte om man skall generera knapparna från en extern datakälla. Med hjälp av en så kallad Repeater (hanteras senare i kompendiet) är det möjligt att dynamiskt generera kontroller utifrån innehållet i exempelvis en databas, vilket är ett troligare scenario än att man för hand skriver in varje kontroll för den aktuella produkten som i exemplet nedan.

Genom att byta händelse till onCommand kan vi ha med ytterligare två attribut i våra knappar, commandName (en sträng) och CommandArgument (kan vara nästan vilket typ av objekt som helst). För att fånga upp den här händelsen måste vi byta ut vår händelsehanterare från en som matchar onclick till en som matchar för onCommand, annars kompilerar inte programmet,

```
<form runat="server">
  <asp:Button runat="server" OnCommand="BuyStuff" Text="Köp" CommandName="12"/>
  <br />
  <asp:Button runat="server" OnCommand="BuyStuff" Text="Köp" CommandName="42"/>
  <br />
  <asp:Button runat="server" OnCommand="BuyStuff" Text="Köp" CommandName="3"/>
  <br />
</form>
```

ASP.NET

```
public partial class ArgumentTest : System.Web.UI.Page
{
    protected void BuyStuff(object sender, CommandEventArgs e)
    {
        int productID = Convert.ToInt32(e.CommandName); // id (12, 42 eller 3)

        // Användaren vill köpa produkten med ID productID,
        // gör något vettigt med den informationen här...
    }
}
```

C# - Notera CommandEventArgs istället för EventArgs.

ASP.NET Uppgift 3-1

Skriv en kalkylator som klarar alla fyra räknesätten och inte kraschar om man dividerar med noll. För att lyckas med detta måste du bygga på en display utifrån de knappar som användaren klickar på, och för enkelhetens skull behöver man inte kunna skriva in siffror i displayen med tangentbordet.

När användaren klickar på ett av räknesätten måste man på något sätt spara ner värdet i displayen för att kunna använda det i uträkningen. Detta görs lämpligen genom att man sparar ner strängen från Text-egenskapen på displayen i ViewState-arrayen och sedan konverterar tillbaka det till en sträng när det behövs i uträkningen.

Vidare behövs också strängar konverteras till siffror vilket avhandlas i bland annat *Rädsla och Avsky* på sidorna 15-16.

Nedanstående exempelkod är bara en början som du kan utgå från om du känner för det (ibland är det lättare att börja från noll också). Notera att knapparna med siffror är bundna till samma händelsehanterare och använder OnCommand och CommandEventArgs för att tala om vilken av knapparna som användaren klickade på.

```
<form runat="server">
  <div>
    <asp:TextBox runat="server" ID="display" ReadOnly="true"/>
    <br />
    <asp:Button runat="server" CommandName="1" Text="1" OnCommand="number"/>
    <asp:Button runat="server" CommandName="2" Text="2" OnCommand="number"/>
    <asp:Button runat="server" CommandName="3" Text="3" OnCommand="number"/>
    <br />
    <asp:Button runat="server" Text=" + " OnClick="plus"/>
    <asp:Button runat="server" Text=" = " OnClick="equals"/>
  </div>
</form>
```

ASP.NET

```
protected void number(object sender, CommandEventArgs e)
{
    display.Text += e.CommandName; // Addera siffran till displayen
}

protected void plus(object sender, EventArgs e)
{
    ViewState["memory"] = display.Text; // Spara ner displayen i kalkylatorns
    display.Text = ""; // "minne" och töm den på text
}

protected void equals(object sender, EventArgs e)
{
    if (ViewState["memory"] != null) // Är det inte null så har vi något sparat
    {
        string memory = (string)ViewState["memory"];
        display.Text = memory + " + " + display.Text;
    }
}
```

C#

Filuppladdning

Filuppladdning görs i ASP.NET med hjälp av kontrollen `<asp:FileUpload>` som renderar sig som en vanlig input med `type="file"` i HTML.

Nedan följer ett exempel, att notera är användandet av `Server.MapPath("~/uploads/")` som expanderar tilde-tecknet till den aktuella webrooten - detta är ett måste då det inte är möjligt att ladda upp till en katalog utan att ange en absolut sökväg.

```
<form runat="server">
  <div>
    <asp:FileUpload id="upload" runat="server"/>
    <asp:Button runat="server" Text="Spara filen" onClick="SaveFile"/>
    <br/>
    <asp:Label runat="server" ID="display" />
  </div>
</form>
```

ASP.NET

```
public partial class upload: System.Web.UI.Page
{
    protected void SaveFile(object sender, EventArgs e)
    {
        if (upload.HasFile)
        {
            try
            {
                // Sparar filen med samma namn i katalogen uploads
                upload.SaveAs(Server.MapPath("~/uploads/" + upload.FileName));

                display.Text = String.Format("Filen {0} sparad ({1} bytes stor)",
                    upload.PostedFile.FileName,
                    upload.PostedFile.ContentLength);
            }
            catch (Exception ex)
            {
                display.Text = "kunde inte spara filen, error: " + ex.Message;
            }
        }
        else
        {
            display.Text = "Ingen fil uppladdad...";
        }
    }
}
```

C#

Något frustrerande är att man inte har direkt åtkomst till filen förrän den har sparats med metoden `SaveAs`, det finns alltså inte som i exempelvis PHP en temporär fil som man kan använda sig av. Istället har man i egenskapen `PostedFile` tillgång till en dataström (`PostedFile.InputStream`) som håller innehållet i filen, vilket kan jämföras med en öppen filpekare.

ASP.NET Uppgift 4-1

Skapa en ny sida som låter användaren ladda upp vilka filer som helst till en katalog, så länge som filerna håller sig under 512Kb i storlek (524288 bytes). Misslyckas uppladdning skall det hanteras på ett sådant sätt att sidan inte kraschar.

ASP.NET Uppgift 4-2

Skapa en ny sida som bara låter användaren ladda upp bilder. Du kan med fördel återanvända koden från föregående exempel.

När en bild är uppladdad skall den visas för användaren tillsammans med filstorleken och bildens proportioner. Använd en `<asp:image>` för att länka till den uppladdade bilden från C#-koden. För att kunna länka till bilden krävs också att den laddas upp i en katalog som är åtkomlig från webbläsaren. Skapa en katalog med namnet "uploads" i det aktuella projektet där filen kan laddas upp om du inte redan gjorde det i uppgift 4-1.

För att kontrollera huruvida en fil är en bild eller inte kan du använda dig av `Bitmap` i namnrymden `System.Drawing` och kapsla in det i en `try/catch` enligt nedanstående exempel (byt ut `stream` mot `InputStream` på den uppladdade filen). Ur `Bitmap`-objektet kan du sedan få ut höjd och bredd som två heltal,



```
// Deklarera variablerna innan så att vi kommer åt dem även utanför try/catch
int height = 0;
int width = 0;

try
{
    System.Drawing.Bitmap image = new System.Drawing.Bitmap(stream);

    // Den här filen är en korrekt bild, plocka ut höjd och bredd
    height = image.Height;
    width = image.Width;
}
catch
{
    // Det här är inte en korrekt bild, hantera det här på något
    // bra sätt (skriv ut ett felmeddelande eller dylikt)
}
```

C#

ASP.NET Uppgift 4-3 (frivillig)

Modifera uppgift 4-2 så att bilden presenteras som en liten bild (en "thumbnail") länkad till den stora bilden om storleken åt något håll överstiger 200 pixlar. Bilden skall justeras proportionsenligt enligt nedanstående välkända algoritm,

```
int height = image.Height;
int width  = image.Width;

if (width > 180 || height > 200)
{
    if (Math.Ceiling(((double)200 / image.Height) * image.Width) < 200)
    {
        height = 200;
        width = (int)Math.Ceiling(((double)200 / image.Height) * image.Width);
    }

    if (Math.Ceiling(((double)200 / image.Width) * image.Height) < 200)
    {
        height = (int)Math.Ceiling(((double)180 / image.Width) * image.Height);
        width = 200;
    }
}

// Spara om bilden med height/width som ny storlek...
```

C#

Det räcker med att du ändrar width och height-attributen på image-kontrollen, men vill du krångla till det för dig och har massa tid över kan du försöka spara ner en kopia av en förminskad bild (*tips*: använd dig av Bitmap-objektet) och använda den för att sedan länka till den stora bilden.

Server tags

Förutom de inbyggda kontrollerna i ASP.NET som alla börjar med ett asp-prefix finns också en speciell tag-syntax som börjar med % följt av ett tecken som representerar någon typ av funktionalitet. Den kanske mest användbara syntaxen är <%= som är en genväg till att skriva ut en textsträng i ASPX-sidan där taggen är placerad,

```
<form runat="server">
  <%= "Hello, world!" %>
</form>
```

ASP.NET - Hello, world!

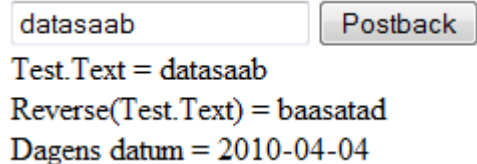
Naturligtvis är exemplet ovan totalt meningslöst, men förutom literära strängar kan vi också skriva ut variabler och returnerade värderna från metoder, vilket kan bespara oss lite onödiga Table-kontroller,

```
<form runat="server">
  <asp:TextBox ID="Test" runat="server"/>
  <asp:button runat="server" text="Postback"/>
  <br />
  Test.Text          = <%= Test.Text %><br />
  Reverse(Test.Text) = <%= Reverse(Test.Text) %><br />
  Dagens datum      = <%= DateTime.Today.ToShortDateString() %><br />
</form>
```

ASP.NET

```
public partial class ServerTagTest : System.Web.UI.Page
{
    protected string Reverse(string s)
    {
        char[] charArray = s.ToCharArray();
        Array.Reverse(charArray);
        return new string(charArray);
    }
}
```

C#



datasaab Postback
Test.Text = datasaab
Reverse(Test.Text) = baasatad
Dagens datum = 2010-04-04

Utskriften från en <%= %> sker i ett relativt sent stadiet av sidans rendering, efter att exempelvis alla händelsehanterare har utförts. Det är därför inte möjligt att blanda utskriften av en <%= %> i en kontroll, det är exempelvis inte tillåtet att skriva något av nedanstående,

```
<form runat="server">
  <asp:TextBox ID="Test" runat="server"/>
  <!-- FEL -->
  <asp:ListBox runat="server">
    <asp:ListItem><%= Test.Text %></asp:ListItem>
  </asp:ListBox>
  <!-- FEL -->
  <asp:textbox runat="server" text="<%= Test.Text %>" />
</form>
```

ASP.NET - sidan kompilarar inte längre.

I ASP.NET 4.0, som i skrivande stund bara är några månader bort från att släppas, rekommenderas det att man istället för <%= %> går över till en ny syntax, <%= %>. Skillnaden lär vara att den nya syntaxen ser till att all utmatning är korrekt HTML-escapad vilket skall omöjliggöra så kallade Cross Site Scripting-attacker (XSS).

MultiView

En `MultiView` är en kontroll som introducerades i ASP.NET 2.0 med syfte att underlätta det väldigt vanliga scenariot att man vill visa en liten del av en sida samtidigt som man döljer annat. En `MultiView` kan bara visa en "vy" i taget och döljer resten, dock utan att kontrollerna i de andra vyerna faktiskt försvinner på sidan (de syns bara inte för användaren). Nedan följer ett exempel på en simpel `MultiView` med tre textboxar där bara en textbox kan visas i taget. För att växla mellan vyerna använder man lämpligen egenskapen `ActiveViewIndex` som representerar ett index från 0 (första vyn) till och med sista, även om det också går att namnge vyerna genom att sätta ID på dem och istället växla mellan dem med `SetActiveView`-metoden.

När en `MultiView` först laddas in har `ActiveViewIndex` ett värde på -1, vilket innebär att ingen vy alls visas.

```
<asp:MultiView runat="server" id="DetailsView">
  <asp:View runat="server">Name <asp:TextBox ID="Name" runat="server"/></asp:View>
  <asp:View runat="server">Phone<asp:TextBox ID="Phone" runat="server"/></asp:View>
  <asp:View runat="server">Email<asp:TextBox ID="Email" runat="server"/></asp:View>
</asp:MultiView>

<asp:Button Text="<" runat="server" ID="PrevButton"
  CommandName="Prev" OnCommand="ChangeView" />

<asp:Button Text=">" runat="server" ID="NextButton"
  CommandName="Next" OnCommand="ChangeView" />

<hr />

Name: <%= Name.Text %><br />
Phone: <%= Phone.Text %><br />
Email: <%= Email.Text %><br />
```

ASP.NET



The screenshot shows a web page with a `MultiView` control. It has three tabs: "Name", "Phone", and "Email". The "Email" tab is currently selected and active, displaying an input field for the email address and two navigation buttons: a left arrow and a right arrow. The other tabs are visible but their content is hidden. The page also displays the text "Name: Floyd Morrison" and "Phone: 042-42 42 42" above the tabs.

```
protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
    {
        DetailsView.ActiveViewIndex = 0; // visa första vyn
        PrevButton.Enabled = false;     // skall inte gå att backa
    }
}

protected void ChangeView(object sender, CommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "Prev": DetailsView.ActiveViewIndex--; break;
        case "Next": DetailsView.ActiveViewIndex++; break;
    }

    if (DetailsView.ActiveViewIndex > 0)
        PrevButton.Enabled = true;
    else
        PrevButton.Enabled = false;     // Gråa ut bakåtknappen om vi har backat
                                         // tillbaka till första vyn.

    if (DetailsView.ActiveViewIndex < DetailsView.Views.Count - 1)
        NextButton.Enabled = true;
    else
        NextButton.Enabled = false;     // Gråa ut framåtknappen om vi är på
                                         // sista vyn
}
```

- Användaren kan fritt hoppa fram och tillbaka i sidans `MultiView` utan att innehållet i textboxarna försvinner.

ASP.NET Uppgift 5-1

Skriv ett frågeformulär liknande illustrationen nedan.

Formuläret skall ha minst fem stycken frågor, alla med bara ett rätt svarsalternativ. När användaren har valt ett av alternativen på en fråga skall nästa fråga direkt visa sig, ända tills sista frågan är besvarad då en summering av antal rätt som användaren har lyckats få ihop skall redovisas på känt manér.

Längst ner under varje fråga skall det finnas en knapp som gör att tidigare svar nollställs och formuläret börjar om på första frågan.

För att åstadkomma ett frågeformulär skall du använda dig av en `<asp:multiview>` där varje fråga ligger i en egen `<asp:view>`.

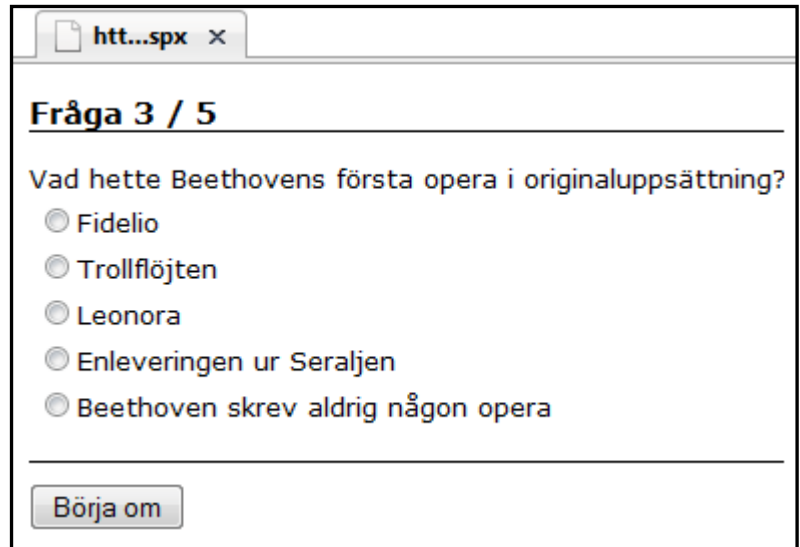
Frågorna skall besvaras i form av en lista med radioknappar som automatiskt postar tillbaka till sidan, använd därför en `<asp:radiobuttonlist>` med `autopostback="true"` (se uppgift 2-1).

När en fråga besvaras måste händelsehanteraren avancera `multiview`-kontrollen till nästa steg samt spara ner antingen om frågan besvarades korrekt, eller, om svaren kontrolleras i slutet, vilket svar som angavs. Du kan använda `ViewState` för att hålla reda på vilka svar eller vilken poäng som användaren har under tiden frågorna besvaras.

Ett annat alternativ är att inte använda `ViewState` alls utan helt enkelt sätta ett specifikt ID på varje `<asp:radiobuttonlist>` och sedan i slutet gå igenom varje lista och se vilka svar som angivits. Hur man löser uppgiften är upp till tycke och smak, men rent generellt gäller naturligtvis att ju mindre kod som är upprepande (måste varje `<asp:radiobuttonlist>` ha en egen händelsehanterare exempelvis?) ju bättre och mer lättläst tenderar programkoden det att bli.

ASP.NET Uppgift 5-2 (frivillig)

Dubblera antal frågor från uppgift 5-1 och skriv om din kod så att programmet slumpmässigt väljer ut fem stycken för användaren att besvara.



The screenshot shows a web browser window with a single tab titled 'htt...spx'. The page content is as follows:

Fråga 3 / 5

Vad hette Beethovens första opera i originaluppsättning?

- Fidelio
- Trollflöjten
- Leonora
- Enleveringen ur Seraljen
- Beethoven skrev aldrig någon opera

Valideringskontroller

För att kontrollera formulärdata kan man i ASP.NET använda så kallade valideringskontroller som kopplas ihop med andra kontroller, ofta textboxar. De inbyggda valideringskontrollerna, förutom `CustomValidator`, kan automatiskt generera JavaScript på klientsidan för att kontrollera formulären utan att behöva posta data tillbaka till servern. Det skall inte vara möjligt att "lura" ASP.NET genom att stänga av JavaScript, en validering på serversidan skall alltid ske och kontrolleras med egenskapen `Page.IsValid`. Flera valideringskontroller kan vara kopplade till en och samma kontroll.

Varje valideringskontroll har två attribut för att meddela eventuella fel till användaren om han eller hon har fyllt i formuläret felaktigt, dels `ErrorMessage` som, precis som det låter, beskriver vad som är fel och dels `Text` som kan användas för att markera vilken kontroll som felet gäller. Valideringskontrollernas attribut `ErrorMessage` används av den speciella kontrollen `ValidationSummary` vars enda uppgift är att ge en sammanfattning över eventuella fel på sidan.

Den absolut lättast valideringskontrollen är troligen `RequiredFieldValidator` som precis som namnet antyder innebär att kontrollen som valideras måste tilldelas ett värde av användaren. Ofta används den här kontrollen tillsammans med någon av de övriga,

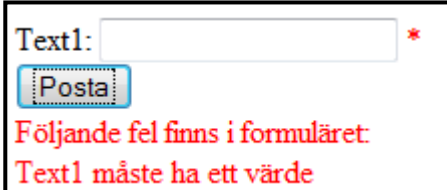
```
<form runat="server">
  Text1:
  <asp:textbox id="text1" runat="server"/>
  <asp:RequiredFieldValidator runat="server" Text="*"
    ControlToValidate="text1"
    ErrorMessage="Text1 måste ha ett värde" />

  <br/>

  <asp:button runat="server" onclick="validateControls"
    text="Posta"/>

  <asp:ValidationSummary runat="server"
    HeaderText="Följande fel finns i formuläret:"
    DisplayMode="List" />
</form>
```

ASP.NET



```
protected void validateControls(object sender, EventArgs e)
{
    if (Page.IsValid)
    {
        // formuläret är OK, gör något här
    }
}
```

C#

Alla webcontrols som är knappar* har den booleska egenskapen `CausesValidation` satt till **true** vilken ASP.NET använder för att automatiskt validera alla kontroller när en sådan postar tillbaka data (och sätter man attributet till **false** sker, naturligtvis, ingen automatisk validering). Används en annan typ av kontroll för att posta tillbaka data måste man manuellt validera sidan, likaså om man i `Page_Load` skulle vilja kontrollera om formuläret är korrekt eller inte, då `Page_Load` som tidigare nämnts anropas av systemet innan händelsehanterarna sparkar igång. Försöker man anropa `Page.IsValid` utan att formuläret faktiskt har blivit validerat kraschar sidan med ett felmeddelande.

* Överkurs: rent tekniskt alla kontroller som implementerar gränssnittet `IButtonControls`.

För att validera sidan manuellt, alltså exempelvis innan en kontroll med CausesValidation satt till **true** har hanterats, måste man anropa metoden Page.Validate(),

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Page.IsPostBack) // Sidan har postats tillbaka till sig själv
    {
        Page.Validate(); // Validera sidans formulär

        if ( ! Page.IsValid)
        {
            // Formuläret är inte OK ifyllt, gör något...
        }
    }
}
```

C#

Förutom RequiredFieldValidator och ValidationSummary kommer ASP.NET med ytterligare ett antal mer eller mindre användbara valideringskontroller,

CompareValidator - Används vanligen för att jämföra en kontrollns värde med en annan men kan också användas för att jämföra en kontroll mot ett fast värde (i så fall används ValueToCompare istället för ControlToCompare),

```
Text1: <asp:textbox id="text1" runat="server"/>
Text2: <asp:textbox id="text2" runat="server"/>
<asp:CompareValidator runat="server" Text="*"
    ControlToValidate="text1"
    ControlToCompare="text2"
    Operator="Equal"
    ErrorMessage="Text1 måste ha samma värde som Text2" />
```

ASP.NET

RangeValidator - Validerar om kontrollen är inom ett visst gränsvärde,

```
Text1:
<asp:textbox id="text1" runat="server"/>
<asp:RangeValidator runat="server" Text="*"
    ControlToValidate="text1"
    MinimumValue="1"
    MaximumValue="10"
    Type="Integer"
    ErrorMessage="Text1 måste innehålla ett heltal mellan 1 och 10"/>
```

ASP.NET

RegularExpressionValidator - Validerar mot ett *regular expression*,

```
Text1:
<asp:textbox id="text1" runat="server"/>
<asp:RegularExpressionValidator runat="server" Text="*"
    ControlToValidate="text1"
    ValidationExpression=".*\?$"
    ErrorMessage="Text1 måste avslutas med ett frågetecken (av någon anledning)" />
```

ASP.NET

Har man speciella behov kan man istället skriva en egendefinerad validator som utför valideringen på serversidan där man naturligtvis kan göra vilka jämförelser som helst. Det är inte tvingande men möjligt att med hjälp av attributet `ClientValidationFunction` anropa en JavaScript-funktion som också validerar på klientsidan,

```
<html>
<script type="text/JavaScript">
  function ValidateTextYesNoJS (sender, args)
  {
    if (args.Value == 'ja' || args.Value == 'nej')
      args.IsValid = true;
    else
      args.IsValid = false;
  }
</script>
<body>
  <form runat="server">
    Är du trött?<br/>
    Svara "ja" eller "nej":

    <asp:textbox runat="server" id="text1"width="20" MaxLength="3"/>
    <asp:button runat="server"onclick="ValidateControls" text="Besvara"/>

    <asp:CustomValidator runat="server" Text="*"
      ControlToValidate="text1"
      ValidateEmptyText="true"
      ClientValidationFunction="ValidateTextYesNoJS"
      OnServerValidate="ValidateTextYesNoCS"
      ErrorMessage="Svara 'ja' eller 'nej'" />

    <asp:button runat="server"
      onclick="ValidateControls"
      text="Besvara"/>

    <hr/>

    <asp:ValidationSummary runat="server" DisplayMode="SingleParagraph" />
  </form>
</body>
</html>
```

JavaScript + ASP.NET

```
protected void ValidateTextYesNoCS(object sender,
  ServerValidateEventArgs e)
{
  if (text1.Text == "ja" || text1.Text == "nej")
    e.IsValid = true;
  else
    e.IsValid = false;
}

protected void ValidateControls(object sender,
  EventArgs e)
{
  if (Page.IsValid)
  {
    if (text1.Text == "ja")
      Label1.Text = "Gå och lägg dig";

    if (text1.Text == "nej")
      Label1.Text = "Drick mindre kaffe";
  }
}
```

C#

Är du trött?
Svara "ja" eller "nej":

Är du trött?
Svara "ja" eller "nej": xyz *
Svara 'ja' eller 'nej'

Är du trött?
Svara "ja" eller "nej": ja
Gå och lägg dig

ASP.NET Uppgift 6-1

Använd en `MultiView` för att skapa ett formulär som stegvis skapar en ny användare för en fiktiv hemsida. Varje steg förutom sista skall ha en knapp som avancerar till nästa del av formuläret.

Steg 1

Första steget skall innehålla textboxar för,

Användarnamn

Valideras på serversidan med hjälp av en `CustomValidator`. Är användarnamnet "leonard", "morrison", eller "floyd" anses det vara upptaget och därför felaktigt.

E-postadress

Valideras med hjälp av en `RegularExpressionValidator` så att den är korrekt skriven.

Lösenord

Valideras med hjälp av en lämplig validerare (undvik `RangeValidator`, antingen `CustomValidator` eller `RegularExpressionValidator` kan användas här) så att det är minst sex tecken långt.

Upprepa Lösenord

Valideras mot lösenordet så att de stämmer överens med hjälp av en `CompareValidator`.

Steg 2

Andra steget skall be om information från användaren,

Kön

presentera varje alternativ som en `RadioButtonList` som valideras med en `RequiredFieldValidator`.

Ålder

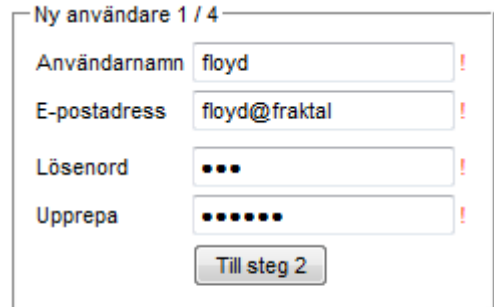
Accepterar en ålder mellan 18 och 100.

Steg 3

Tredje steget skall vara en summering av vad användaren har skrivit och en knapp som låter användaren gå tillbaka till första delen av formuläret om han eller hon vill ändra sina uppgifter.

Steg 4

Fjärde och sista steget skall bara skriva ut "Ny användare skapad, välkommen *användarnamn*".



Användarnamnet är upptaget
Felaktig e-postadress
För kort lösenord
Lösenorden stämmer inte



Välj hur du ser på dig själv i spegeln
Du måste vara över 18 år

Master Pages

Introducerade med ASP.NET 2.0, *Master Pages* (någon bra vedertagen svensk översättning saknas, men Mastersida är väl det närmaste vi kommer) är ett sätt att i ASP.NET få en enhetlig design utan att använda sig av små inkluderingsfiler som stegvis bygger upp sidan. En Mastersida fungerar istället tvärt om, den utgör stommen och sedan är det upp till den aktuella sidan man faktiskt är inne på att lägga sig på rätt ställe och manipulera Mastersidans utseende. Det må låta krångligt, och visst är det lite konfunderande i början, men när man väl förstår konstruktionen kan det lätt kännas svårt att gå tillbaka till det "vanliga" sättet med inkluderingsfiler efteråt.

En Mastersida är i stort sätt som vilken webform som helst, med skillnaden att den slutar med *.master* istället för *.aspx* och har tillgång till en speciell ASP.NET-kontroll, `ContentPlaceholder`, som används för att markera var innehållen på sidan skall finnas. En Mastersida kan och har också ofta så flera olika `ContentPlaceholder` beroende på hur avancerad layouten är. Det är inte möjligt att direkt surfa till en Mastersida med en webbläsare utan en användare måste alltid gå igenom en "vanlig" sida som i sin tur använder sig av Mastersidan.

Nedan ett exempel på en Mastersida (`Default.master`) som innehåller en `ContentPlaceholder` och en separat sida (`Content.aspx`) som använder Mastersidan.

```
<%@ Master Language="C#" CodeBehind="Default.master.cs" Inherits="Default" %>
<html>
  <body>
    <form runat="server">
      MasterLabel: <asp:Label ID="MasterLabel" runat="server"/>
      <div style="border: solid 1px #000; padding:5px; margin:5px;">
        <asp:ContentPlaceholder ID="TheContent" runat="server">
          </asp:ContentPlaceholder>
      </div>
    </form>
  </body>
</html>
```

ASP.NET - Default.master

MasterLabel:

ContentLabel:

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

C# - Default.master.cs

```
<%@ Page Language="C#" CodeBehind="Content.aspx.cs" Inherits="Content"
    MasterPageFile="~/Default.Master"%>
<asp:Content ContentPlaceHolderID="TheContent" runat="server">
  ContentLabel: <asp:Label runat="server" id="ContentLabel"/>
</asp:Content>
```

ASP.NET - Content.aspx

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

C# - Content.aspx.cs

Vad kan vi dra för slutsatser av föregående sida, förutom att det användes väldigt mycket kod för att åstadkomma ytterst lite? Först kan vi konstatera att Mastersidan innehåller en <form> medan Content-sidan inte gör det. Mastersidan ärver från klassen `MasterPage` medan Content-sidan, precis som vanligt, ärver från `Page`. Ingen av sidorna gör något i sin C#-kod än.

Det som säger att Content-sidan använder sig av `Default.Master` är `MasterPageFile`-attributet i siddirektivet. Så fort en sida använder sig av det attributet måste allt innehåll på sidan ligga inuti en eller flera content-kontroller. Dessa kontroller måste i sin tur ha ett `ContentPlaceHolderID`-attribut som skall stämma överens med ett ID på en `ContentPlaceHolder` på Mastersidan. En hel del att hålla reda på. I det här fallet finns det bara en `ContentPlaceHolder`, med det inte allt för fantasifulla ID "TheContent". När användaren surfar in på `Content.aspx` kompilerar ASP.NET ihop sidorna till slutresultatet att "ContentLabel"-texten hamnar inuti den <div> som finns på Mastersidan.

Fördelen på en sådan här enkel sida är naturligtvis försumbar, för att inte säga kontraproduktiv, men ju fler sidor man har och ju mer avancerat projektet blir ju mer kommer man uppskatta att slippa återupprepa mycket av samma kod; även med inkluderingsfiler måste man ofta ha något typ av skellett på varje enskild sida som därefter placerar in filerna på rätt ställe, något man helt kan delegera till Mastersidan i ASP.NET.

En situation som uppstår ganska snart är att man vill ändra på saker i Mastersidan från en innehållsida (i brist på bättre ord) - kanske vill man dölja fält i layouten eller uppdatera någon kontroll. Det här är inte så enkelt som det låter, då man inte automatiskt har direkt åtkomst till Mastersidan från en innehållsida såvida man inte uttryckligen definierar vilken datatyp (klass) som Mastersidan som man använder är av. Att en sida inte automatiskt kan veta vilken datatyp som Mastersidan är av kan tyckas som aningen fånigt, men har att göra mer den nya kompileringsmodellen som introducerades i ASP.NET 2.0, vilket visserligen inte är någon ursäkt men en förklaring.

För att komma åt Mastersidan från en innehållssida används egenskapen `Master`, men utan att ange vilken datatyp som Mastersidan är av kommer `Master`-egenskapen på innehållssidan peka på alla Mastersidors basklass (alltså `System.Web.UI.MasterPage`), men vi vill i exemplet på förra sidan att den skall vara av typen `Default` (mastersidans klass). Skulle vi försöka komma åt den `Label` som finns på Mastersidan i dagsläget på det här sättet,

```
public partial class Content : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Master.MasterLabel.Text = "Hello";
    }
}
```

C#

...skulle vi dels inte få någon intellisense på `MasterLabel`, vilket i sig nästan garanterat är ett tecken på att vi har gjort fel, och när vi kompilerar programmet skulle vi få felet (naturligtvis, eller hur?),

'System.Web.UI.MasterPage' does not contain a definition for 'MasterLabel'

Det finns två sätt att lösa det här problemet, antingen kan vi göra en *typecast* till rätt datatyp,

```
public partial class Content : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        ((Default)Master).MasterLabel.Text = "Hello"
    }
}
```

C# - Content.aspx.cs

...vilket fungerar men ser väldigt fult ut. Naturligtvis finns det en bättre lösning, nämligen att under @Page-direktivet introducera ett @MasterType-direktiv som talar om, antingen via sökväg (VirtualPath) eller med klassnamn (TypeName), vilken datatyp som mastersidan är av,

```
<%@ Page Language="C#" CodeBehind="Content.aspx.cs" Inherits="Content"
    MasterPageFile="~/Default.Master"%>
<%@ MasterType VirtualPath="~/Default.Master" %>

<asp:Content ContentPlaceHolderID="TheContent" runat="server">
    ContentLabel: <asp:Label runat="server" id="ContentLabel"/>
</asp:Content>
```

ASP.NET - Content.aspx

Oavsett vilken av varianterna ovan man väljer kommer man *fortfarande* inte få någon intellisense och sidan kommer *fortfarande* inte kompilera, vilket lätt kan få en att tro att man inte kommer åt mastersidan än trots allt. Men det gör man, kompileringensfelet är helt enkelt bara ett annat nu,

'Default.MasterLabel' is inaccessible due to its protection level

Felet ligger i att alla kontroller i ASP.NET 2.0 och senare genereras i en dold klass som instansvariabler med modifieraren **protected**, som innebär att variabeln bara är tillgänglig för den egna klassen och klasser som ärver från den. Innehållssidan använder sig visserligen av Mastersidan, men den ärver inte av klassen (den ärver av Page) och det finns ingenting* vi kan göra för att ändra modifieraren till **public** eller få innehållssidan att ärva från mastersidan.

Lösningen står att finna i avsnittet om *datahiding* i *Rädsla och Avsky* (sidan 22), vi måste från Mastersidan *exponera* de kontroller som vi vill att andra sidor skall kunna komma åt. För att exponera en kontroll skapar man i Mastersidan lämpligen en simpel publik egenskap för detta (se också exemplet på sidan 26 i *Rädsla och Avsky*). Då den variabeln som vi vill exponera redan heter MasterLabel får vi antingen döpa egenskapen annorlunda eller döpa om variabeln, det vill säga kontrollens ID. Som nämns i *Rädsla och Avsky* är det vanligt att man låter variabeln ha en liten begynnelsebokstav och egenskapen stor, vilket vi kommer göra här, men vi sätter också ett däckat minustecken framför variabeln för att vara extra tydliga på hur vi skiljer dem åt.

* Det där är egentligen inte alls sant; rent tekniskt så kan du flytta variabeln från den autogenererade filen Default.Master.designer.cs in i CodeBehind-filen Default.Master.cs och ändra **protected** till **public** (och det fungerar), men editorn kommer inte uppdatera variabelnamnet om du byter namn på kontrollens ID, utan istället skapa en ny i den autogenererade filen. Rekommenderas inte.

```

<%@ Master Language="C#" CodeBehind="Default.master.cs" Inherits="Default" %>
<html>
  <body>
    <form runat="server">
      MasterLabel: <asp:Label ID="_masterLabel" runat="server"/>
      <div style="border: solid 1px #000; padding:5px; margin:5px;">
        <asp:ContentPlaceHolder ID="TheContent" runat="server">

          </asp:ContentPlaceHolder>
        </div>
      </form>
    </body>
  </html>

```

ASP.NET - Default.master

MasterLabel: Hello

ContentLabel: World

```

public partial class Default : System.Web.UI.MasterPage
{
    public Label MasterLabel
    {
        get { return _masterLabel; }
        set { _masterLabel = value; }
    }

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

```

C# - Default.master.cs

```

<%@ Page Language="C#" CodeBehind="Content.aspx.cs" Inherits="Content"
    MasterPageFile="~/Default.Master"%>
<%@ MasterType VirtualPath="~/Default.Master" %>

<asp:Content ContentPlaceHolderID="TheContent" runat="server">
  ContentLabel: <asp:Label runat="server" id="ContentLabel"/>
</asp:Content>

```

ASP.NET - Content.aspx

```

public partial class Content : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        Master.MasterLabel.Text = "Hello";
        ContentLabel.Text = "World";
    }
}

```

C# - Content.aspx.cs

Även om det inte framgår av exemplet ovan får vi också intelligens på `MasterLabel` direkt efter att vi skrivit "Master." i `Page_Load`. I det här fallet så exponerar vi *hela* `Label`-kontrollen, vilket man visserligen oftast också vill, men man hade lika gärna kunnat välja att bara exponera `Text`-egenskapen och då fått ändra datatypen till **string** (och i `Content.aspx` hade man fått skriva `Master.MasterLabel = "Hello"` istället),

```

public string MasterLabel
{
    get { return _masterLabel.Text; }
    set { _masterLabel.Text = value; }
}

```

C# - Mastersidan exponerar nu bara sträng-egenskapen och inte hela kontrollen.

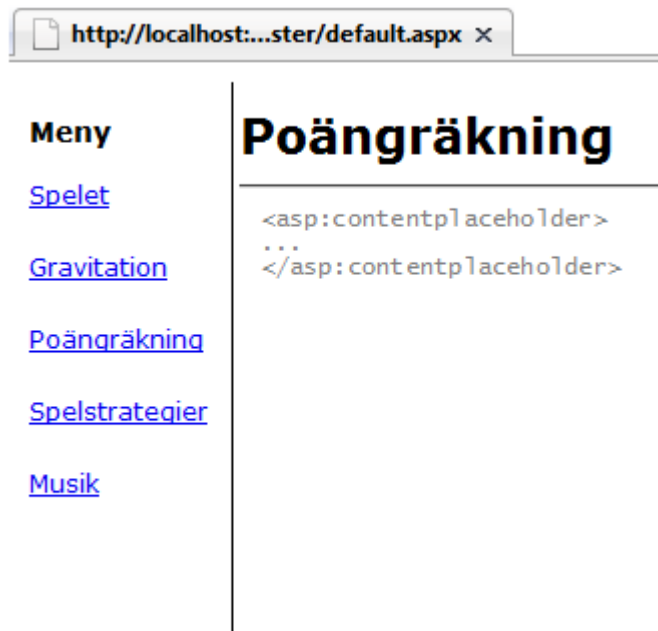
ASP.NET Uppgift 7-1

i den uppgiften kan du återanvända den texten du använde i uppgift 2-1, men har du tappat bort den kan du bara gå till wikipedia och kopiera från en ny lämplig artikel.

Skapa en mastersida som innehåller en meny med länkar och en överskrift i fet stil, enligt exemplet till höger.

I det tomma utrymmet under överskriften lägger du in en `contentPlaceholder` som sedan skall fyllas i från de sidorna som meny-länkar till.

Vidare skall också överskriften ändras till det aktuella menyvalet, och detta skall göras från samma sida som textinnehållet ligger i samtidigt som själva asp-kontrollen med överskriften fortfarande skall befinna sig i Master-sidan.



ASP.NET Uppgift 7-2 (frivillig)

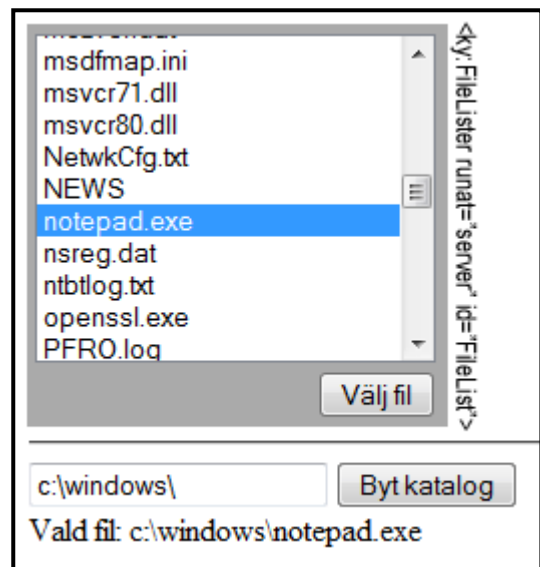
Försök lösa uppgift 7-1 så att det aktuella menyvalet på något sätt blir markerat och inte går att klicka på igen när det är valt. Det här är inte fullt så enkelt som det låter utan kräver nog en del klurande.

Användarkontroller

I ASP.NET finns det två typer av kontroller som en användare kan skapa, *User Controls* (användarkontroller) och *Custom Controls*. Skillnaden ligger bland annat i att en Custom Control inte har någon direkt design utan måste rendera sig genom att i C#-koden skriva HTML "för hand", en Custom Control är den typen av kontroller som ASP.NET använder sig av och är aningen klurigare att skriva än användarkontroller. En fördel med Custom Controls är att de kan kompileras ner till DLL-filer och enkelt inkluderas eller säljas till tredjepart. Användarkontroller, vilket är vad vi skall titta på, är relativt enkla att skriva och går att jämföra med en inkluderings sida. Innan mastersidor introducerades i ASP.NET fyllde användarkontroller ofta rollen att dela upp layouten i återanvändbara delar, medan de idag kanske mest används för enstaka återkommande funktioner (som exempelvis menyer) som bara delas av några enstaka sidor och därför blir bökiga om man skall ha in dem på Mastersidorna också.

För att skapa en ny användarkontroll i Visual Web Developer väljer man istället för ett nytt webform alternativet *Web User Control*. Användarkontroller skiljer sig på så vis att de har en annan filändelse (.ascx), ärver från UserControl istället för Page och har ett @Control-direktiv istället för ett @Page-direktiv. Annars liknar och fungerar de i stort sätt som vanligt, förutom att de inte har någon <form runat="server"> då de är tänkta att läggas in på andra sidor.

I det här exemplet skall vi skapa en användarkontroll enligt illustrationen till höger som består av en listbox och en knapp. Listboxen skall användas för att lista alla filer i en given katalog och knappen skall i slutändan gå att svara på från den sidan som kontrollen ligger på. Sidorna som använder kontrollen skall också kunna välja vilken katalog som skall listas genom att sätta en egenskap på den från sin C#-kod.



```
<%@ Control Language="C#" CodeBehind="FileLister.ascx.cs" Inherits="FileLister" %>
<table style="background-color:#aaa;">
<tr><td>
    <asp:ListBox runat="server" ID="FileListBox" Rows="10" width="200">
    </asp:ListBox>
</td></tr>
<tr><td align="right">
    <asp:Button runat="server" Text="Välj fil" />
</td></tr>
</table>
```

ASP.NET - FileLister.ascx

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

C# - FileLister.ascx.cs

Precis som för mastersidor är det inte möjligt att direkt provköra en användarkontroll genom att ladda in den direkt i webbläsaren, utan man måste alltid ha en sida som använder sig av den. För att använda en användarkontroll på en sida måste den *registreras*, antingen i konfigurationsfilen `web.config`, eller som vi kommer göra här, individuellt på varje sida som skall använda den med hjälp av ett `@Register`-direktiv,

```
<%@ Page Language="C#" CodeBehind="TestUserControl.aspx.cs"
    Inherits="TestUserControl" %>
<%@ Register Src="~/FileLISTER.ascx" TagPrefix="ky" TagName="FileLISTER" %>

<html>
<body>
    <form runat="server">
        <ky:FileLISTER runat="server" ID="FileList" />
        <hr />
        <asp:TextBox runat="server" ID="DirectoryToUse" />
        <asp:Button runat="server" Text="Byt katalog" />
        <br />
        Vald fil: <asp:Label runat="server" ID="FileLabel"/>
    </form>
</body>
</html>
```

ASP.NET - TestUserControl.aspx

För att registrera vår nya totalt meningslösa `FileLISTER`-kontroll måste vi skriva in sökvägen och hitta på ett `TagPrefix` och ett `TagName`. *Vanligtvis* ger man `TagName` samma namn som vad kontrollen heter, medan `TagPrefix` brukar vara en förkortning för företaget eller projektnamnet som kontrollen kommer från eller ingår i. Det finns inga definitiva regler utan man kan använda vilka namn som helst när man registrerar en kontroll på sidan.

Med detta har vi all layout på plats och kan rendera kontrollen i en testsida, vad som saknas är C#-koden som gör att något faktiskt händer.

Till att börja med kan vi lägga in en metod i användarkontrollen som faktiskt fyller listboxen med filnamnen från en katalog. För detta måste vi använda oss av klassen `DirectoryInfo` i namnrymden `System.IO` som Visual Web Developer vanligen inte brukar lägga till högst upp i C#-filen automatiskt.

Koden på nästa sida är relativt enkel att förstå sig på, först töms listboxen på alla eventuella rader, sedan, om vi har med oss en sökväg som argument, skapar vi ett nytt `DirectoryInfo`-objekt och går igenom alla filer med en **foreach**. Det må tyckas lite lustigt att använda metoden `File.Exists` för att ta reda på om ett filnamn är en fil eller en katalog, men det är faktiskt så man gör. För varje fil som finns lägger vi till en ny rad med namnet i listboxen och till sist, om vi har lagt till några rader i listboxen, så markerar vi den översta i listan.



```

using System.IO;

public partial class FileLister : System.Web.UI.UserControl
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if ( ! IsPostBack)
            UpdateList(@"C:\");
    }

    private void UpdateList(string path)
    {
        FileListBox.Items.Clear();

        if (String.IsNullOrEmpty(path))
            return;

        DirectoryInfo dir = new DirectoryInfo(path);

        foreach (FileInfo file in dir.GetFiles())
        {
            if (File.Exists(file.FullName))
                FileListBox.Items.Add(file.Name);
        }

        if (FileListBox.Items.Count > 0)
            FileListBox.SelectedIndex = 0;
    }
}

```

C# - FileLister.ascx.cs

Problemet med ovanstående kod är naturligtvis att vi bara använder oss av C:\ och att det inte är möjligt att från sidorna som använder kontrollen att själva ändra sökväg. Vad som behövs är ett attribut på kontrollen för detta, så att användaren istället kan skriva exempelvis,

```
<ky:FileLister runat="server" ID="FileList" Path="C:\SPEL\DOOM2\"/>
```

För att skapa attribut i kontroller använder man egenskaper, vars syntax vi förhoppningsvis har lite koll på vid det här laget. I det här fallet uppstår ett nytt problem dock, vi vill att vår kontroll skall "komma ihåg" vilket värde den har i Path mellan PostBacks, så att man först skall kunna ändra sökvägen och därefter klicka på ett filnamn och få tillbaka hela filnamnet inklusive sökvägen. För att komma ihåg värdet måste vi manuellt spara ner det i ViewState,

```

public partial class FileLister : System.Web.UI.UserControl
{
    public string Path
    {
        get { return (string)ViewState["path"]; }
        set { ViewState["path"] = value;
              UpdateList(value);
            }
    }
    ...
}

```

C# - Varje gång som någon ändrar Path sparar vi ner värdet i ViewState och uppdaterar listan med filer.

Hade vi inte använt ViewState här hade alltså kontrollen INTE kommit ihåg sitt Path-attribut mellan PostBacks.

Det bör noteras att alla kontroller har sin "egen" ViewState, det vill säga att även om vi har flera FileLister på samma sida eller om sidan som använder kontrollen också försöker använda ViewState["path"] så är de helt separerade från varandra.

Användaren kan nu både sätta attributet Path direkt på kontrollen och uppdatera sökvägen från C#-koden,

```
...
<form runat="server">
  <ky:FileLister runat="server" ID="FileList" Path="C:\tmp\"/>
  <hr />
  <asp:TextBox runat="server" ID="DirectoryToUse" />
  <asp:Button runat="server" Text="Byt katalog" OnClick="ChangeDirectory"/>
...
```

ASP.NET - TestUserControl.aspx

```
public partial class TestUserControl : System.Web.UI.Page
{
    protected void ChangeDirectory(object sender, EventArgs e)
    {
        FileList.Path = DirectoryToUse.Text;
    }
}
```

C# - TestUserControl.aspx.cs

Bara två saker saknas nu, först måste vi ha ett *read-only*-attribut (egenskap som saknar **set**) som talar om vilken fil som är vald för tillfället. Det är enkelt, vi har ju ett ID på listboxen i kontrollen och kan plocka fram vilken rad som är vald, och vi vet vilken sökväg som användaren har satt eftersom vi har sparat ner den i ViewState.

```
public partial class FileLister : System.Web.UI.UserControl
{
    public string SelectedFile
    {
        get { return (string)ViewState["path"] + FileListBox.SelectedValue; }
    }
    ...
}
```

C# - FileLister.ascx.cs

Det sista och kanske svåraste är att låta sidan som använder kontrollen lyssna på om någon fil blir vald, det vill säga att om användaren klickar på knappen "Välj fil" vill vi att sidan som använder kontrollen skall kunna ha en händelsehanterare som fångar upp att en fil är vald. För detta måste vi skapa ett **event**, vilket är aningen överkurs eftersom **event** och dess storebror **delegate** inte finns med i *Rädsla och Avsky*, men vi använder det här ändå eftersom det annars blir en ofullständig kontroll.

Den färdiga koden deklarerar ett **event** och implementerar händelsehanteraren `ButtonClicked` för knappen med "Välj fil". Om någon "lyssnar" på händelsen `FileSelected` kommer den inte vara **null** och vi kan aktivera vårt **event** genom att skicka oss själva som `sender` och ett tomt `EventArgs` (`EventArgs.Empty` är ett i ASP.NET optimerat sätt att säga `new EventArgs()`, vilket också hade fungerat).

```
<%@ Control Language="C#" CodeBehind="FileLister.ascx.cs" Inherits="FileLister" %>
<table style="background-color:#aaa;">
<tr><td>
    <asp:ListBox runat="server" ID="FileListBox" Rows="10" width="200">
    </asp:ListBox>
</td></tr>
<tr><td align="right">
    <asp:Button Text="Välj fil" runat="server" OnClick="ButtonClicked"/>
</td></tr>
</table>
```

ASP.NET - FileLister.ascx

```
using System.IO;

public partial class FileLister : System.Web.UI.UserControl
{
    public event EventHandler FileSelected;

    public string SelectedFile
    {
        get { return (string)ViewState["path"] + FileListBox.SelectedValue; }
    }

    public string Path
    {
        get { return (string)ViewState["path"]; }
        set {
            ViewState["path"] = value;
            UpdateList(value);
        }
    }

    protected void ButtonClicked(object sender, EventArgs e)
    {
        if (FileSelected != null)
            FileSelected(this, EventArgs.Empty);
    }

    private void UpdateList(string path)
    {
        FileListBox.Items.Clear();

        if (String.IsNullOrEmpty(path))
            return;

        DirectoryInfo dir = new DirectoryInfo(path);

        foreach (FileInfo file in dir.GetFiles())
        {
            if (File.Exists(file.FullName))
                FileListBox.Items.Add(file.Name);
        }

        if (FileListBox.Items.Count > 0)
            FileListBox.SelectedIndex = 0;
    }
}
```

C# - FileLister.ascx.cs

I den färdiga koden för sidan som använder kontrollen lyssnar vi på `onFileSelected` (prefixet "On" läggs dit av systemet automatiskt), det vill säga vi knyter en händelsehanterare till den, där allt vi gör är att uppdatera `Label`-kontrollen med vilken fil som är vald.

```
<%@ Page Language="C#" CodeBehind="TestUserControl.aspx.cs"
      Inherits="TestUserControl" %>
<%@ Register Src="~/FileList.ascx" TagPrefix="ky" TagName="FileList" %>

<html>
<body>
  <form runat="server">
    <ky:FileList runat="server" ID="FileList" onFileSelected="FileChosen"/>
    <hr />
    <asp:TextBox runat="server" ID="DirectoryToUse" />
    <asp:Button runat="server" Text="Byt katalog" onClick="ChangeDirectory"/>
    <br />
    Vald fil: <asp:Label runat="server" ID="FileLabel" />
  </form>
</body>
</html>
```

ASP.NET - TestUserControl.aspx

```
public partial class TestUserControl : System.Web.UI.Page
{
    protected void FileChosen(object sender, EventArgs e)
    {
        FileLabel.Text = FileList.SelectedFile;
    }

    protected void ChangeDirectory(object sender, EventArgs e)
    {
        FileList.Path = DirectoryToUse.Text;
    }
}
```

C# - TestUserControl.aspx.cs

ASP.NET Uppgift 8-1

För att lösa den här uppgiften måste du ta reda på hur man läser textfiler med C#. Detta står inte i Rädsla och Avsky utan precis som i verkligheten får du klura ut det på egen hand (det vill säga fråga Google).

Skriv en användarkontroll med namnet `IncludeText`, vars uppgift det är att inkludera en textfil på sidan. Kontrollen skall ha följande egenskaper:

`Filename` - sökväg och filnamn till den fil som skall inkluderas. Är den här egenskapen en tom sträng skall kontrollen inte visa någon utdata överhuvudtaget.

`Header` - överskrift, i fet text med ett streck under. Är den här egenskapen en tom sträng skall varken överskrift eller strecket visas.

`FontSize` - Skall bara kunna ha värdena `Small`, `Normal`, `Large` och sätter storleken på texten till den filen som skall skrivas ut.

Exempel

```
<test:IncludeText
  runat="server"
  filename="C:\stranger.txt"
  header="On Stranger Tides"
  fontsize="Normal" />
```

För att få `FontSize` att vara begränsad till bara tre värden måste du använda dig av en egendefinierad `enum`.

För att sätta storleken på texten kan du använda dig av CSS-attributet `font-size` och sedan sätta en lämplig textstorlek. För att göra detta på en kontroll från C# kan du exempelvis skriva,

```
kontrollen.Style["font-style"] = "14px";
```

On Stranger Tides

I was sorting through some boxes today and I came across my copy of Tim Power's *On Stranger Tides*, which I read in the late 80's and was the inspiration for *Monkey Island*. Some people believe the inspiration for *Monkey Island* came from the *Pirates of the Caribbean* ride - probably because I said it several times during interviews - but that was really just for the ambiance.

ASP.NET Uppgift 8-2 (frivillig)

Skapa en ny sida som använder sig av en tom `IncludeText` från uppgift 8-1. Lägg till en listbox på sidan med namnet på tre filer. När användaren väljer en av filerna från listboxen skall C#-koden fånga upp händelsen och säga till kontrollen vilken fil som är vald och vilken header som skall användas. Problemet är att händelsehanteraren för listboxen kommer fångas upp först *efter* `Page_Load` i `IncludeText`, vilket innebär att om det är där all kod ligger i kontrollen kommer sidan förbli tom. Försök klura ut hur du skall få `IncludeText` att fungera så att man från en händelsehanterare på en sida kan sätta vilken fil som skall användas och få den utskrivna. Du kommer vara tvungen att ändra på koden i kontrollen och troligen få konsultera Google.

Autentisering

Autentisering i ASP.NET sköts vanligen till stor del av ASP.NET själv med hjälp av direktiv i konfigurationsfilen `web.config`, som placeras i den katalog vars filer kräver autentisering. Autentiseringen slår även igenom på eventuella underkataloger, vilket gör att man kan lösenordsskydda hela sin applikation genom att använda sig av den `web.config` som finns i rooten.

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <allow users="floyd, leonard"/> <!-- tillåt floyd och leonard -->
      <deny users="*/> <!-- neka alla andra -->
    </authorization>
  </system.web>
</configuration>
```

web.config - Tillåter bara användarnamnen *floyd* och *leonard*.

För att bara tillåta autentiserade användare och stänga ute alla andra kan vi plocka bort `<allow>` helt och ändra `<deny users="*/>` till `<deny users="?">`,

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authorization>
      <deny users="?" /> <!-- neka alla okända -->
    </authorization>
  </system.web>
</configuration>
```

web.config - Tillåter vem som helst bara de är inloggade.

Efter detta måste vi ange vilken autentiseringsmetod som skall gälla och vilken URL som användaren skall hamna på när han eller hon försöker komma åt en lösenordsskyddad sida. Detta görs också i en `web.config` och *måste*, såvida man inte använder så kallade *virtuella kataloger* i IIS, definieras i den `web.config` som ligger i rooten på applikationen. I normala fall är det så kallad *forms-authentication* som skall användas, vilket innebär att vi själva vill validera inloggningen (vanligen mot en databas med användare),

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <authentication mode="Forms">
      <forms loginUrl="~/login.aspx" />
    </authentication>
  </system.web>
</configuration>
```

web.config

Den URL som är definierad som `loginUrl` är undantagen från autentiseringen även om den ligger i en "skyddad" katalog.

För att användaren skall kunna logga in skapar man sedan en login.aspx, eller vilket namn man nu angav i web.config, där man skapar lämpliga kontroller som exempelvis två textfält för användarnamn och lösenord och en knapp som skall validera inloggningen,

```
Username: <asp:textbox runat="server" id="username"/>
Password: <asp:textbox runat="server" id="password"/>
         <asp:button runat="server" onclick="Login"/>
```

ASP.NET

```
protected void Login(object sender, EventArgs e)
{
    if (username.Text == "floyd" && password.Text == "fraktal" ||
        username.Text == "leonard" && password.Text == "skalbagge")
    {
        // Loggar in användaren
        FormsAuthentication.RedirectFromLoginPage(username.Text, false);
    }
}
```

C#

Den statiska metoden RedirectFromLoginPage sätter användarnamnet i användarens session och skickar personen vidare till den sida som gjorde att användaren kom till sidan med loginformuläret från början. Har användaren inte kommit från någon sida skickas han eller hon till index-sidan istället. Klassen FormsAuthentication ligger i namnrymden System.Web.Security vilken man vanligen måste lägga till en referens till manuellt överst i C#-filen.

För att på en sida testa om och med vilket användarnamn en användaren är inloggad med samt i så fall synliggöra en knapp som kan användas för att logga ut,

```
<asp:label runat="server" id="testLabel"/>
<asp:button runat="server" id="logoutButton" text="Logout" onclick="Logout"/>
```

ASP.NET

```
using System.Web.Security;

protected void Page_Load(object sender, EventArgs e)
{
    if (User.Identity.IsAuthenticated)
    {
        testLabel.Text = "Inloggad som " + User.Identity.Name;
        logoutButton.Visible = true;
    }
    else
    {
        testLabel.Text = "Inte inloggad..";
        logoutButton.Visible = false;
    }
}

protected void Logout(object sender, EventArgs e)
{
    FormsAuthentication.SignOut(); // Loggar ut användaren
    Response.Redirect("goodbye.aspx"); // Flyttar användaren till en annan sida
}
```

C#

Det finns också ett flertal kontroller i ASP.NET vars funktionalitet är baserade på huruvida användaren är inloggad eller inte. En av de mer användbara är `<asp:Loginview>` (ny i ASP.NET 2.0) som kan visa olika *templates*, ungefär som en `<asp:MultiView>`, fast istället byggd runt användarens identitet. Med hjälp av den kontrollen skulle vi kunna ta bort all kod i `Page_Load` på föregående sida och få samma resultat genom att ändra vår ASP.NET-kod till,

```
<asp:LoginView runat="server">
  <AnonymousTemplate>
    Inte inloggad...
  </AnonymousTemplate>

  <LoggedInTemplate>
    Inloggad som <%= User.Identity.Name %>
    <asp:button runat="server" text="Logout" onclick="Logout"/>
  </LoggedInTemplate>
</asp:LoginView>
```

ASP.NET

Att via C# komma åt kontroller i en `<asp:Loginview>` är dock aningen knepigt, då kontroller skapade i en av dess templates inte existerar när man kompilerar sidan. Det är därför inte möjligt att direkt komma åt dem genom att skriva kontrollens ID som vi är vana; istället får man för hand plocka ut kontrollerna med hjälp av `FindControl` som dynamiskt kan hämta kontroller från den template som är aktuell (fortfarande med hjälp av kontrollens ID dock). Metoden `FindControl` är i sig inte unik för just `<asp:Loginview>` utan finns på alla kontroller. Nackdelen med att använda den är att kompilatorn inte kan fånga upp om man skulle skriva ett felaktigt kontrollnamn, och det är därför rekommenderat att bara använda metoden om situationen så absolut kräver.

Nedan följer ett exempel på hur vi i en `AnonymousTemplate` kan använda textboxar för användarnamn och lösenord och sedan hämta ut de två kontrollerna dynamiskt när användaren klickar på Login-knappen. För att komma åt `FindControl` på vår `<asp:Loginview>` måste vi sätta ett ID på den och eftersom `FindControl` inte i förväg kan veta vilken typ av kontroll den returnerar måste vi för hand konvertera kontrollen till sin rätta datatyp, antingen med (`TextBox`) eller med hjälp av det reserverade ordet **as**. Nedanstående exempel använder båda varianterna.

```
<asp:LoginView runat="server" id="LoginStatusView">
  <AnonymousTemplate>
    Username: <asp:textbox id="username" runat="server">
    Password: <asp:textbox id="password" runat="server">
      <asp:button runat="server" text="Login" onclick="Login"/>
    </AnonymousTemplate>
</asp:LoginView>
```

ASP.NET

```
protected void Login(object sender, EventArgs e)
{
    TextBox username = (TextBox)LoginStatusView.FindControl("username");
    TextBox password = LoginStatusView.FindControl("password") as TextBox;

    if (username.Text == "floyd" && password.Text == "fraktal" ||
        username.Text == "leonard" && password.Text == "skalbagge")
    {
        FormsAuthentication.RedirectFromLoginPage(username.Text, false);
    }
}
```

C#

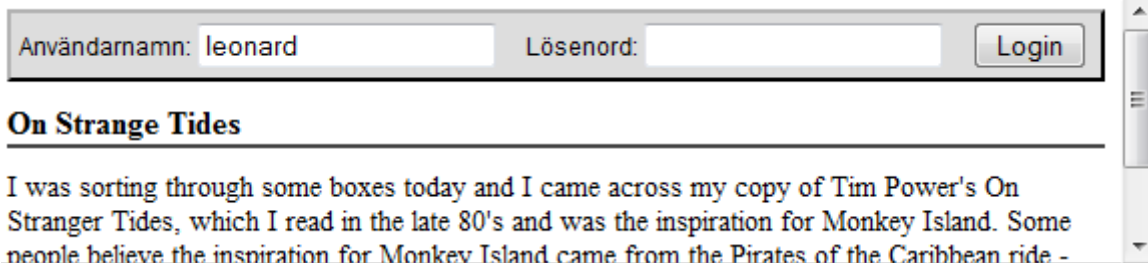
ASP.NET Uppgift 9-1

I den här uppgiften återanvänds användarkontrollen från uppgift 8-1 som inkluderar en textfil på en sida, du måste med andra ord göra den uppgiften först. Du får också fråga Google om hur man skriver till textfiler i C#.

Skapa en mastersida som alla sidor skall använda sig av. Mastersidan skall innehålla ett login-formulär om man inte är inloggad och en logout-knapp om man är det. Under login-formuläret skall en ContentPlaceHolder-kontroll finnas där andra sidor kommer skriva in sitt innehåll.

Skapa en default.aspx som använder sig av användarkontrollen från uppgift 8-1 och helt enkelt skriver in en textfil till master-sidan, det blir med andra ord inte mer komplicerad än,

```
<asp:Content ContentPlaceHolderID="Content" runat="server">
  <test:IncludeText runat="server"
    Filename="C:\stranger.txt" Header="On Stranger Tides" FontSize="Normal" />
</asp:Content>
```



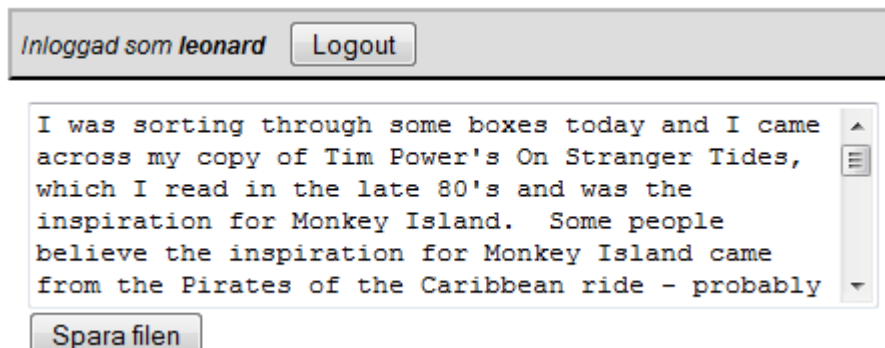
The screenshot shows a web page with a login form at the top. The form has two input fields: "Användarnamn:" with the value "leonard" and "Lösenord:". To the right of the password field is a "Login" button. Below the form is a section titled "On Strange Tides" with a horizontal line underneath. The text below the line reads: "I was sorting through some boxes today and I came across my copy of Tim Power's On Stranger Tides, which I read in the late 80's and was the inspiration for Monkey Island. Some people believe the inspiration for Monkey Island came from the Pirates of the Caribbean ride -".

Skriv en händelsehanterare för login-knappen i mastersidan och hårdkoda in en användare och ett lösenord i C#-koden. Misslyckas man med att logga in kan detta presenteras genom att man skriver ut ett felmeddelande under login-formuläret. När man loggar ut skall man automatiskt förflyttas tillbaka till default.aspx i rooten på applikationen.

Skapa en underkatalog kallad "admin" dit användaren inte har åtkomst såvida han eller hon inte är autentiserad. I den underkatalogen skapar du en default.aspx med en multiline-textbox,

```
<asp:TextBox runat="server" TextMode="MultiLine" Rows="5" Columns="50" Text=""/>
```

Under textboxen lägger du till en knapp med texten "Spara filen". När användaren som inloggad besöker sidan skall textboxen fyllas med texten från den textfil som används (du får hårdkoda in vilken, det går inte att direkt komma åt IncludeText-kontrollen på den andra default.aspx-sidan). Knappen "Spara filen" skall naturligtvis spara eventuella ändringar i textfilen.



The screenshot shows a web page where the user is logged in. At the top, it says "Inloggad som leonard" next to a "Logout" button. Below this is a multiline text area containing the text: "I was sorting through some boxes today and I came across my copy of Tim Power's On Stranger Tides, which I read in the late 80's and was the inspiration for Monkey Island. Some people believe the inspiration for Monkey Island came from the Pirates of the Caribbean ride - probably". Below the text area is a "Spara filen" button.

DataBinding

Konceptet med *DataBinding* är centralt i ASP.NET och ett måste att lära sig om man skall jobba med databaser eller andra externa källor med information. På det hela stora handlar det om att från en datakälla extrahera information och tilldela denna till olika kontroller på sidan. Rent praktiskt kan *DataBinding* även användas för att automatiskt tilldela data från en existerande kontroll till en annan, men detta användningsområde ligger utanför den här textens avhandling och måste nog generellt anses som väldigt ovanligt förekommande.

I sitt kanske absolut simplaste utförande kan man binda ihop en `<asp:DropDownList>` enligt nedanstående exempel med en sträng-array,

```
<asp:DropDownList runat="server" ID="countries"></asp:DropDownList>
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        string[] data = { "Sverige", "Norge", "Danmark", "Finland" };
        countries.DataSource = data;
        countries.DataBind();
    }
}
```

C#

```
<select name="countries" id="countries">
  <option value="Sverige">Sverige</option>
  <option value="Norge">Norge</option>
  <option value="Danmark">Danmark</option>
  <option value="Finland">Finland</option>
</select>
```

Resultat

För att binda ihop data med kontrollen måste man, som i exemplet ovan, först ange vilken datakälla som skall användas i egenskapen `DataSource` och sedan anropa `DataBind()` (glömmer man anropa `DataBind()` binds inte datan och inget händer). Det är också viktigt att notera att kontrollen tappar all tidigare information när man binder data till den, vilket gör att man för det mesta vill undvika att binda om kontrollen varje gång data postas tillbaka till sidan.

Så gott som all typ av data* som det går att iterera genom kan användas som datakälla i ASP.NET. I ovanstående tämligen grundläggande exempel användes en endimensionerad array vilket skapade en `<asp:dropdown>` som använder samma data (i det här fallet namnet på ett land) som både "Text" och "Value". Detta är visserligen inte så konstigt då arrayen inte hade någon annan information, men det är naturligtvis inte alltid som man har sådan "tur" att den datakällan man vill använda är så simpel som ovan.

* Överkurs: rent tekniskt alla objekt av klasser som implementerar `IEnumerable`, `ICollection` eller `IListSource`.

I ett mer avancerat exempel har vi en kollektion av typen `List<T>` som innehåller objekt från en egendefinierad klass. ASP.NET kan använda kollektionen för att binda data mot vår `<asp:dropdown>` men kan i förväg inte veta vilken egenskap på objekten som skall användas. För listkontroller finns därför de två egenskaperna `DataTextField` och `DataValueField` som används för att uttryckligen definiera vilka egenskaper på objektet som skall användas som `Text` respektive `Value`,

```
<asp:DropDownList runat="server" ID="replicants"></asp:DropDownList>
```

ASP.NET

```
using System.Collections.Generic;
public partial class DataBindTest : System.Web.UI.Page
{
    class Replicant
    {
        public string ID { get; set; }
        public string Name { get; set; }

        public Replicant(string id, string name)
        {
            this.ID = id;
            this.Name = name;
        }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!IsPostBack)
        {
            var data = new List<Replicant>()
            {
                new Replicant("N6MAA10816", "Roy Batty"),
                new Replicant("N6MAC41717", "Leon Kowalski"),
                new Replicant("N6FAB21416", "Pris")
            };

            replicants.DataSource = data;
            replicants.DataValueField = "ID";
            replicants.DataTextField = "Name";
            replicants.DataBind();
        }
    }
}
```

C#

```
<select name="replicants" id="replicants">
  <option value="N6MAA10816">Roy Batty</option>
  <option value="N6MAC41717">Leon Kowalski</option>
  <option value="N6FAB21416">Pris</option>
</select>
```

Resultat

Det kan vara noterbart att en vanlig sträng används för att definiera egenskaperna som skall användas. Skulle man skriva fel och ange en egenskap som inte finns kraschar sidan med ett `HttpException` när `DataBind()` anropas på kontrollen, man får med andra ord ingen hjälp av kompilatorn i det här fallet.

Databaser

Att binda en kontroll till data från en databas är i teorin samma konstruktion som att binda den till en array eller en kollektion - det är samma DataSource-egenskap och samma DataBind-metod som tidigare, men rent praktiskt krävs det viss merjobb eftersom man dels måste skapa en anslutning till databasen och dels måste använda ytterligare några klasser för att faktiskt få ut någon data som sedan kan bindas.

ConnectionString

För att kunna ansluta till en databasserver med ASP.NET används en *ConnectionString*, en vanlig textsträng som håller reda på inloggning, typ av databasserver, och vilken databas som skall användas. Vanligen vill man kunna komma åt denna information från mer än sida varför det finns en egen sektion för denna sträng i web.config som man kan komma åt överallt,

```
<connectionStrings>
  <add name="Connection" connectionString="Connection-strängen"
</connectionStrings>
```

web.config

För att komma åt strängen från C# använder man sig av ConfigurationManager i namnrymden System.Configuration enligt följande,

```
ConfigurationManager.ConnectionStrings["Connection"].ConnectionString;
```

C#

DataSet och SqlDataAdapter

Det finns i ASP.NET olika lösningar för att sedan rent praktiskt prata med databasen, vilket tyvärr lätt gör det hela aningen svåröverskådligt. Tills vidare skall vi koncentrera oss på att använda klassen DataSet för att binda data från en SELECT-query. Ett DataSet är egentligen ganska komplext men kan i det här fallet för enkelhetens skull beskrivas som data från en SELECT-query samlad i ett objekt som man sedan binder mot.

Tyvärr kan vi inte i ett svep fylla vårt DataSet med data från en databas utan vi måste använda oss av ett mellanlager, SqlDataAdapter, som ligger mellan DataSet och databaskopplingen. En SqlDataAdapter tar vanligen två argument: SELECT-queryn och Connection-strängen,

```
using System.Configuraiton; // Behövs för ConfigurationManager
using System.Data; // Behövs för DataSet
using System.Data.SqlClient // Behövs för SqlDataAdapter

...
// Skapar ett nytt, tomt dataset
DataSet ds = new DataSet();

// Skapar en databaskoppling och gör en SELECT...
SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT...",
    ConfigurationManager.ConnectionStrings["Connection"].ConnectionString);

// Fyller dataset med data från SELECT-satsen
adapter.Fill(ds);
```

C#

Repeater

I det tidigare exemplet användes en listkontroll för att presentera data från en datakälla, men det är naturligtvis inte alltid eller ens vidare ofta som det räcker med en enskild kontroll på det viset, speciellt inte om datan kommer från en databas. I många språk där layout och programkod ligger blandade med varandra löser man ofta detta genom att helt enkelt gå igenom ("loopa") datan en rad från databasen i taget och sedan presentera det genom att för varje rad skriva ut ett HTML-fragment. Tankegången är ungefär densamma i ASP.NET, även om tillvägagångssättet blir aningen knepigare.

Det bör påpekas att det också finns ett antal olika fördefinierade kontroller för att på ett snabbt sätt kunna presentera data från en databas, exempelvis `<asp:GridView>` och `<asp:ListView>`, men dessa färdiga kontroller för med sig att de bara har ett sätt att presentera datan på och är därför, om än smidiga i vissa fall, väldigt begränsande och totalt oanvändbara om man själv vill ha full kontroll över vilken HTML som skall produceras.

För att få ut data på sidan skall vi istället använda oss av kontrollen `<asp:Repeater>` som fungerar ungefär som scenariot med att loopa igenom data och skriva ut HTML för varje rad, och är den databundna kontrollen som får mest användning av programmerare som inte vill få så mycket "hjälp" av systemet utan själva vill kunna finlira med sin HTML-utskrift.

En Repeater-kontroll består av ett antal *templates* som automatiskt blir anropade när data binds till kontrollen. De är,

<HeaderTemplate>

Om den finns definierad anropas den precis innan den första raden av data skall skrivas ut. Ett typiskt exempel vore att i en `<HeaderTemplate>` skriva en öppnande `<table>` om man vill presentera sin data i en tabell.

<FooterTemplate>

Och motsatsen, anropas när det inte längre finns några rader att skriva ut, en lämplig kandidat att skriva här vore en avslutande `</table>`.

<ItemTemplate>

En `<ItemTemplate>` anropas varje gång en ny rad med data är redo att skrivas ut och måste därför alltid vara med om man vill få ut något på sidan. I en `<ItemTemplate>` existerar ett speciellt objekt, `Container.DataItem`, som innehåller den aktuella datan (exempelvis en rad från databasen). För att faktiskt få ut datan på sidan när den kommer från en databas måste man brottas med metoden `DataBinder.Eval` som placeras i en speciell databindningssyntax, `<%# %>`, och tar `Container.DataItem` och namnet på den kolumnen vars data man vill ha ut som argument. Komplicerat? Absolut, men med hjälp av ett komplett exempel på nästa sidan kanske ändå syntaxen blir lite lättare att förstå sig på.

<AlternatingItemTemplate>

Fungerar som en `<ItemTemplate>` och blir, om den finns definierad, anropad varannan gång, vilket gör det enkelt att exempelvis skriva ut varannan rad med en annan typ av layout.

<SeperatorTemplate>

Anropas om den finns definierad mellan varje `<ItemTemplate>` och `<AlternatingItemTemplate>`.

Exempel

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <connectionStrings>
      <add name="Connection"
          connectionString="Data Source=.\\SQLEXPRESS;
                          Initial Catalog=webshop;
                          Integrated Security=True"
      />
    </connectionStrings>
  </system.web>
</configuration>
```

web.config

```
protected void Page_Load(object sender, EventArgs e)
{
    if (!IsPostBack)
    {
        DataSet ds = new DataSet();

        SqlDataAdapter adapter = new SqlDataAdapter(
            "SELECT TOP 10 ProductID, ProductName FROM Products",
            ConfigurationManager.ConnectionStrings["Connection"].ConnectionString);

        adapter.Fill(ds);
        repeater.DataSource = ds;
        repeater.DataBind();
    }
}
```

C#

```
<asp:repeater runat="server" id="repeater">
  <HeaderTemplate>
    <table>
      <tr>
        <th>ID</th>
        <th>Namn</th>
      </tr>
    </HeaderTemplate>
    <ItemTemplate>
      <tr>
        <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
        <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
      </tr>
    </ItemTemplate>
    <FooterTemplate>
      </table>
    </FooterTemplate>
</asp:repeater>
```

ASP.NET

```
<table>
  <tr>
    <th>ID</th>
    <th>Namn</th>
  </tr>
  <tr>
    <td>15</td>
    <td>Best of Leonard Skalbagge</td>
  </tr>
  <tr>
    <td>73</td>
    <td>Zombie cats, real or fiction?</td>
  </tr>
</table>
```

Resultat

Varför DataBinder.Eval behövs

Det är lätt att se `DataBinder.Eval` som något som man alltid måste ha med i sin repeater utan att reflektera över *varför* den behövs, vilket lätt leder till att man också uppfattar syntaxen för databindningar som onödigt komplex. Sanningen är den att `DataBinder.Eval` alls inte behövs så länge som den datan som binds i repeatern visas korrekt när den omvandlas till en sträng,

```
protected void On_Load(object sender, EventArgs e)
{
    int[] tmp = { 7, 5, 8, 42 };
    repeater.DataSource = tmp;
    repeater.DataBind();
}
```

C#

```
<asp:repeater runat="server" id="repeater">
  <ItemTemplate>
    <## Container.DataItem %>
  </ItemTemplate>
</asp:repeater>
```

7 5 8 42

ASP.NET

Exemplet ovan fungerar därför att vi binder mot en array med heltal istället för att använda data från en databas. Varje tal i arrayen går igenom ett i taget och omvandlas till en sträng - det fungerar på samma sätt som om vi i ett konsolprogram hade skrivit,

```
int[] tmp = { 7, 5, 8, 42 };
foreach (int i in tmp)
{
    Console.WriteLine(i); // skriver ut 7 5 8 42
}
```

C#

Anledningen till varför `DataBinder.Eval` måste användas när datan kommer från en databas är för att `Container.DataItem` i ett sådant fall är en mer komplex datatyp - den innehåller en rad från databasen där *flera olika kolumner med data* kan förekomma, och med `DataBinder.Eval` måste vi uttryckligen ange *vilken* av de olika kolumnerna som skall användas - detta även om vår SELECT-query bara returnerar en kolumn.

Så vad händer om vi "glömmer" att använda `DataBinder.Eval` på en rad från databasen? Eftersom `Container.DataItem` inte kan representera sig som en sträng faller den tillbaka till standarden för alla objekt som inte själva definierar hur de skall se ut som strängar, nämligen klassens namn och namnrymd, i det här fallet `System.Data.DataRowView`.

Nedanstående exempel illustrerar skillnaden med data från en fiktiv databas med en ID-kolumn,

```
<asp:repeater runat="server" id="repeater">
  <ItemTemplate>
    <## Container.DataItem %> ...
    <## DataBinder.Eval(Container.DataItem, "ID") %>
    <br/>
  </ItemTemplate>
</asp:repeater>
```

System.Data.DataRowView ... 3
System.Data.DataRowView ... 5
System.Data.DataRowView ... 9

ASP.NET

Att använda data i en template

I det kompletta exemplet några sidor tillbaka användes bara statisk data, det vill säga datan presenterades utan att användaren kunde göra något med den. Låt säga att man skulle vilja generera en knapp bredvid varje rad som användaren kan använda för att markera att han eller hon vill köpa, ta bort, eller på annat sätt manipulera datan som representeras av just den raden. Till att börja med kan vi lägga till en knapp i vår `<ItemTemplate>` enligt känd modell,

```
<ItemTemplate>
  <tr>
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
    <td><asp:button id="SelectButton" text="Select Product" /> </td>
  </tr>
</ItemTemplate>
```

ASP.NET

Nu blir det lite lurigt - en kontroll inuti en "template" fungerar annorlunda jämfört med en knapp utanför. Som tidigare nämnts i avsnittet om autentisering där kontrollen `LoginView` diskuterades existerar inte knappen i en template när sidan kompileras, utan den skapas istället *dynamiskt* först när templatens renderas på sidan. Detta är logiskt om man tänker efter; systemet kan inte i förväg veta hur många knappar som skall genereras, och detta gör det omöjligt att direkt från sidans C#-kod komma åt knappen med hjälp av dess ID som vi är vana eftersom knappen inte existerar än. Vidare är det inte heller möjligt att veta vilken produkt som blir vald i exemplet ovan; skulle vi lägga till en vanlig `onClick`-metod på knappen måste vi på något sätt skicka med ytterligare information om vilken av knapparna som användes och vilket produkt-ID som den representerar. Lyckligtvis har vi en lösning på det problemet med `onCommand` som kan skicka med ytterligare argument till händelsehantlaren, exempelvis aktuellt produkt-ID,

```
<ItemTemplate>
  <tr>
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
    <td>
      <asp:button id="SelectButton" text="Select Product"
        OnCommand="ProductSelected"
        CommandName='<%=# DataBinder.Eval(Container.DataItem, "ProductID") %>' />
    </td>
  </tr>
</ItemTemplate>
```

ASP.NET

```
protected void ProductSelected(object sender, CommandEventArgs e)
{
  int ProductID = Convert.ToInt32(e.CommandName);
  // Gör något med ProductID...
}
```

C#

Det ser lite grötigt ut men fungerar, `CommandName` i ASP.NET-koden använder sig alltså av den databundna syntaxen `<%=# %>` för att plocka ut aktuellt ID och vi kan på så vis fånga upp vilket ID som knappen är tänkt att representera. Också att notera är hur vi måste använda apostrof istället för citattecken runt `<%=# %>` för att inte krocka med citattecknet som omsluter "ProductID".

ASP.NET Uppgift 10-1

Skapa en simpel filhanterare över den lokala hårddisken enligt illustrationen. För detta använder du två stycken Repeater-kontroller; en för katalogerna och en för filerna. För att binda datan skapar du först en instans av DirectoryInfo (från System.IO) som tar en sökväg som argument, och som datakälla på dina två Repeater använder du sedan GetDirectories() respektive GetFiles() på det instansierade objektet. Glöm inte att anropa DataBind() efter du tilldelat datakällan.

\\Program Files\\Adobe\\Adobe Photoshop CS3\\Help\\



..



[additional how to content](#)



[images](#)



[howto.dat](#)



[Photoshop CS3 Help.pdf](#)

För att hålla reda på var användaren befinner sig kan du ha en variabel i ViewState som innehåller aktuell sökväg.

Länkarna i din repeater för kataloger är lämpligen LinkButton-kontroller som skickar med katalognamnet i CommandName som lätt kan fångas upp med en händelsehanterare för OnCommand. LinkButton-kontroller fungerar som Button-kontroller, enda skillnaden är hur de renderar sig. Bygg sedan på den aktuella sökvägen och bind om dina repeater för att "byta katalog".

ASP.NET Uppgift 10-2

Som du snabbt kommer märka så returnerar inte GetDirectories katalogerna "." och ".." , där den senare används för att gå tillbaka i kataloghierarkin. Har du gjort färdigt uppgift 10-1 har du med andra ord en filhanterare där du bara kan gå in i kataloger, men aldrig tillbaka. Implementera, på valfritt sätt, något som gör att ".." renderas överst (som i illustrationen ovan) eller något annat smart system för att kunna backa tillbaka ner i kataloghierarkin.

ASP.NET Uppgift 10-3 (Frivillig)

Skriv en händelsehanterare för vad som händer om man klickar på en fil istället för en katalog. Är det en textfil eller en bild skall innehållet dumpas ut i webbläsaren (med rätt Content-Type), annars skall en "Save As..."-rutan komma upp för användaren så att filen kan laddas ner.

Avancerad <ItemTemplate>

Än så länge har vår <ItemTemplate> varit väldigt simpel i sin layout, vi har helt enkelt utgått från att all data från databasen skall presenteras på samma sätt hela tiden. Vår <asp:repeater> har som nämnts en speciell <AlternatingItemTemplate> som, om den används, anropas varannan gång för att ändra utseende på varannan post som skall skrivas ut. Den existerar då det är något som man ofta vill göra, exempelvis för att markera varannan rad med en annan bakgrundsfärg för att öka läsbarheten,

```
<ItemTemplate>
  <tr style="background-color:#fff;">
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
  </tr>
</ItemTemplate>
<AlternatingItemTemplate>
  <tr style="background-color:#ccc;">
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
    <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
  </tr>
</AlternatingItemTemplate>
```

ASP.NET - Varannan post vit, varannan grå

Det här räcker dock inte om man vill göra något som ligger utanför ramarna, som att ändra layouten på var tredje rad istället, eller visa en annan typ av layout för alla produkter som har ett nersatt pris, eller något annat baserat på den aktuella datan. I ett språk där programkod och layout inte är lika åtskilt som i ASP.NET är det här ofta ett ganska enkelt problem att lösa då man med en simpel **if**-sats ofta kan bestämma vilken typ av layout som skall spottas ut, medan det i ASP.NET blir lite krångligare.

Direkt kan vi konstatera att <AlternatingItemTemplate> är oduglig för allt annat än att rendera sig varannan gång, därför kan vi plocka bort den och bara hålla oss inom vår <ItemTemplate> istället. Tankegången blir att först definiera de olika layouterna som skall användas inom denna enda template och sedan baserat på något villkor bestämma vilken av dem som skall visas.

Vi kan börja med att lösa problemet med att vilja använda en specifik layout baserad på vilken rad som visas, exempelvis då att var tredje rad istället för som ovan varannan skall visas med en grå bakgrund. Det här är egentligen ett relativt simpelt problem att lösa då allt vi egentligen behöver göra är att för var tredje rad stoppa in en CSS-regel på <tr>. För att komma åt vår tr gör vi den till en HtmlControl (rent formellt en HtmlTableRow egentligen) och tilldelar den ett id.

```
<asp:Repeater runat="server">
  <HeaderTemplate>
    <table>
  </HeaderTemplate>
  <ItemTemplate>
    <tr runat="server" id="row">
      <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
      <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
    </tr>
  </ItemTemplate>
  <FooterTemplate>
    </table>
  </FooterTemplate >
</asp:Repeater>
```

ASP.NET

Vad vi nu vill åstadkomma är att varje gång som våran `<ItemTemplate>` är beredd att genereras skall en händelsehanterare anropas och vi skall därifrån bestämma om det skall in en CSS-regel eller inte på raden. En Repeater-kontroll har en sådan händelse i `OnItemDataBound` och det enda som vi behöver göra är att skriva in ett metodnamn och definiera metoden i våran kod precis som vanligt. Nästan i alla fall, händelsehanteraren för `OnItemDataBound` tar som sitt andra argument inte `EventArgs` utan `RepeaterItemEventArgs`, vilken innehåller information om den aktuella raden, bland annat vilken rad i ordningen som är på väg att skrivas ut - vilket är precis vad vi kommer att använda oss av,

```
<asp:Repeater runat="server" OnItemDataBound="SelectLayout">
  <ItemTemplate>
    <tr runat="server" id="row">
      <td><%=# DataBinder.Eval(Container.DataItem, "ProductID") %></td>
      <td><%=# DataBinder.Eval(Container.DataItem, "ProductName") %></td>
    </tr>
  </ItemTemplate>
</asp:Repeater>
```

ASP.NET

```
protected void SelectLayout(object sender, RepeaterItemEventArgs e)
{
    // Anropars när varje gång som en template renderar sig
}
```

C#

Det här är lite lurigt, våran `SelectLayout` kan nämligen komma att anropas fler gånger än en gång för varje post som kommer från databasen då `OnItemDataBound` utförs även för de andra template-typerna. Har vi exempelvis en `<HeaderTemplate>` kommer `OnItemDataBound` först anropas för den innan den kommer till den verkliga datan som vi är intresserad av. För att lösa detta får vi med hjälp av egenskapen `Item` i `RepeaterItemEventArgs` kontrollera vilken typ av template som `OnItemDataBound` faktiskt har reagerat på. Vidare är det också så att våran `<ItemTemplate>` automatiskt kommer bli en `<AlternatingItemTemplate>` varannan gång även om det inte finns någon definierad. Så, med en `if`-sats i våran `SelectLayout` kan vi börja med att kontrollera vilken typ av template det är som gäller och om vi skall göra något,

```
protected void SelectLayout(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        // SelectLayout har anropats av en ItemTemplate, gör något här...
    }
}
```

Nästa problem är att ta reda på hur många rader som har genererats för att kunna lista ut om vi skall visa raden med eller utan en CSS-regel som sätter en ny bakgrundsfärg. Praktiskt nog slipper vi hålla reda på detta själva (även om vi teoretiskt skulle kunna genom att deklarera en instansvariabel och addera den med 1 för varje anrop till `SelectLayout`), då våran `e.Item` innehåller egenskapen `ItemIndex` som börjar på 0 för den första raden och sedan ökar med 1 för varje rad som har skrivits ut.

För att sedan ta reda på om det är en "tredje rad", som då skall skrivas ut med grå bakgrund, använder vi den förhoppningsvis inte obekanta %-operatören (modulus) för att ta reda på om `e.Item.ItemIndex` är jämt delbart med 3,

```
protected void SelectLayout(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        if (e.ItemIndex % 3 == 0)
        {
            // Var tredje rad...
        }
    }
}
```

C#

Det sista problemet att lösa är att vi nu måste komma åt den tabellraden där CSS-regeln faktiskt skall in, och eftersom den ligger i en `<ItemTemplate>` existerar den inte när sidan blir kompilerad utan skapas dynamiskt för varje rad som skall skrivas ut. Att därför direkt försöka skriva exempelvis,

```
row.Style["background-color"] = "#aaa";
```

skulle ge oss ett kompilatorfel då ingen sådan kontroll existerar förrän datan är bunden. Eftersom vi vet att raden kommer existera när datan blir bunden kan vi dynamiskt plocka fram den med hjälp av metoden `FindControl` på vårt `e.Item`. Det här är ganska så klurigt rent tekniskt men vi kan leva i glad ovisshet över att det bara fungerar. Metoden `FindControl` returnerar tillbaka den kontrollen som matchar det ID som används som argument, men kan (naturligtvis) inte i förväg veta vilken typ av kontroll som den skall returnera. Istället får vi för hand konvertera det objekt som den returnerar till en `HtmlControl` då vi vet vilken typ det är, enligt följande,

```
protected void SelectLayout(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        if (e.ItemIndex % 3 == 0)
        {
            // Plocka ut tr för den aktuella raden
            HtmlControl row = e.Item.FindControl("row") as HtmlControl;

            // sätt bakgrundsfärgen på den
            row.Style["background-color"] = "#aaa";
        }
    }
}
```

C#

Det finns inget som hindrar oss från att skicka in ett ID till `FindControl` som inte existerar, eller att vi försöker konvertera om det objektet den returnerar till något felaktigt. Båda scenariorna leder till ett `null`-objekt vilket, om man försöker använda det, resulterar i ett `NullReferenceException` och en kraschad sida.

Att basera sin layout på den aktuella datan som är bunden är ytterligare ett steg som också är allt annat än uppenbart enkelt. Låt säga som ett banalt exempel att vi vill ha en annan fontstorlek på alla produkter som börjar med bokstaven A. Vi kan återanvända vår kunskap från föregående äventyr där vi använde en enda <ItemTemplate> var i vi placerade all vår layout och sedan skev händelsehanterare för OnItemDataBound på Repeater-kontrollen,

```
<asp:Repeater runat="server" OnItemDataBound="SelectLayout">
  <ItemTemplate>
    <asp:label id="ProductLabel" runat="server"
      text='<## DataBinder.Eval(Container.DataItem, "ProductName") %>' /><br/>
    </ItemTemplate>
</asp:repeater>
```

ASP.NET

```
protected void SelectLayout(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        // Ändra storlek om produktens namn börjar med "A" här
    }
}
```

C#

Det enda egentliga problemet att lösa nu är hur man får ut den aktuella bundna datan i sin händelsehanterare. Eftersom en <asp:Repeater> teoretiskt kan binda sig mot annan data än det som hämtas ur en databas finns det bara en generisk (i brist på bättre svenskt ordval) egenskap i e.Item som representerar den aktuella raden, och precis som i fallet med FindControl måste vi själva säga att datan kommer från en rad i ett Dataset, en så kallad DataRowView,

```
DataRowView datarow = e.Item.DataItem as DataRowView
```

Variabeln datarow innehåller nu den aktuella raden i databasen - samtliga kolumner från SELECT-satsen (även de som inte skrivs ut på sidan) - och vi kan därifrån plocka ut rätt kolumn genom att använda indexer-syntax (klamrar). Tyvärr, åter igen, kan inte systemet i förväg veta vilken typ av kolumn det handlar om så vi får konvertera den till rätt datatyp för hand, i det här fallet till en **string** så att vi kan kontrollera första bokstaven. Hela pannkakan blir då något i stil med följande,

```
protected void SelectLayout(object sender, RepeaterItemEventArgs e)
{
    if (e.Item.ItemType == ListItemType.Item ||
        e.Item.ItemType == ListItemType.AlternatingItem)
    {
        DataRowView datarow = e.Item.DataItem as DataRowView;
        string name = Convert.ToString(datarow["ProductName"]);

        if (name.StartsWith("A"))
        {
            // Plocka ut aktuella "labeln" och ändra storleken
            Label ProductName = e.Item.FindControl("ProductName") as Label;
            ProductName.Style.Add("font-size", "25px");
        }
    }
}
```

C#

ASP.NET Uppgift 11-1

Återanvänd filhanteraren från uppgift 10. Lägg till en `onItemDataBound` på repeatern för listningen av filnamn. Istället för att visa en enda ikon för alla typer av filer, ändra så att bilder och textfiler (baserat på filändelsen) har egna ikoner.

ASP.NET Uppgift 11-2


Samma filhanterare som uppgiften ovan - lägg till en `onItemDataBound` på Repeater-kontrollen för kataloger och skriv efter varje katalognamn ut hur många filer och kataloger det finns i den. Du kan få fram hur många filer och kataloger som finns genom att använda,

```
var dir = new DirectoryInfo(sökväg);  
var total = dir.GetDirectories().Length + dir.GetFiles().Length;
```

Glöm inte att det måste vara *hela* sökvägen, du kan inte bara använda katalognamnet från `e.Item.DataItem` som sökväg utan måste först slå ihop den med den sökväg som för tillfället används i filhanteraren som aktuell katalog.

ASP.NET Uppgift 11-3

Skapa en ny databas i SQL Server med en tabell kallad "Products". Tabellen skall se ut enligt följande,

	Column Name	Data Type	Allow Nulls
	ID	int	<input type="checkbox"/>
	Name	varchar(50)	<input type="checkbox"/>
	Price	int	<input type="checkbox"/>

Lägg till 10 stycken påhittade produkter med valfritt pris (för enkelhetens skull är priset heltal, utan ören). Skriv en ny sida som på ett mer eller mindre tjugigt sätt listar produkterna i alfabetisk ordning med prisuppgift. Alla produkter som har ett pris under 100:- är att betrakta som extra billiga och de priserna skall skrivas ut i fet röd text.

ASP.NET Uppgift 11-4 (Frivillig)

Skapa en listning av produkterna så som illustrationen visar, det vill säga att den växlar mellan grå och vit bakgrund var tredje rad.

Produkter till salu

Blandade 2600 från åren 1987-1989	150:-
Blue Box från 1970	370:-
Eric S. Raymonds utgångna vapenlicens	50:-
IBM Model M, utan svenska tecken	140:-
Opublicerad manual till Copland OS	190:-
Tac-2, passar allt med DE-9-kontakt	75:-
Trasig kompass med "JWZ" ingraverat	10:-

UPDATE, DELETE, INSERT, etc

Att använda SQL-kommandon som inte returnerar något, det vill säga allt som inte är en SELECT, är en relativt smärtfri process och görs med hjälp av `ExecuteNonQuery()` på ett `SqlCommand`-objekt från namnrymden `System.Data.SqlClient`. Tillvägagångssättet består i tur och ordning av följande moment:

- 1 Skapa en anslutning mot databasen (`SqlConnection`)
- 2 Skapa ett SQL-kommando (`SqlCommand`)
- 3 Öppna databaskopplingen
- 4 Utför SQL-kommandot
- 5 Stäng databaskopplingen

Det är viktigt att man verkligen alltid stänger ner databaskopplingen när man är klar då ASP.NET/SQL-Server till skillnad mot vissa andra system inte gör det automatiskt. För att göra detta 100% säkert måste man trassla in sin kod i en `try`-sats och istället för (eller tillsammans med) en `catch`-sats måste det i slutet finnas en `finally`-sats som ser till att databasen alltid kommer att stängas ner korrekt. Att göra detta varje gång man vill utföra en SQL-query är, som det låter, lite bökit, och det finns därför en speciell kortare syntax för sådana här tillfällen som går ut på att man återanvänder ordet `using` och kapslar in det objekt som när det är färdig använt måste avslutas korrekt. Att använda `using` istället för `try/finally` är så kallad syntaktiskt socker och tillför ingenting annat än att källkoden ser bättre ut,

```
string query = "DELETE FROM Products WHERE ProductID = 5";
string constr = ConfigurationManager.ConnectionStrings["DB"].ConnectionString;

using (SqlConnection connection = new SqlConnection(constr)) // 1
{
    SqlCommand command = new SqlCommand(query, connection); // 2
    connection.Open(); // 3
    command.ExecuteNonQuery(); // 4
} // 5
```

C# - Korrekt sätt att utföra en vanlig SQL-query som inte är en SELECT

Ovanstående kan alternativt skrivas såhär,

```
SqlConnection connection = null;

try
{
    connection = new SqlConnection(constr); // 1
    SqlCommand command = new SqlCommand(query, connection); // 2
    command.Connection.Open(); // 3
    command.ExecuteNonQuery(); // 4
}
finally
{
    if (connection != null) // 5
        connection.Close();
}
```

C# - Också korrekt, fast lite bökitare

Den uppmärksamma läsaren kanske noterar att i tidigare exempel när vi har jobbat med `SqlDataAdapter` har vi inte använt oss av en `SqlConnection`, men ändå lyckats ansluta mot databasen. Detta beror på att när `SqlDataAdapter` tar en SQL-Query (i ren text) och en `ConnectionString` som argument kan den automatiskt öppna och stänga anslutningen på egen hand.

Att undvika SQL-injections

SQL Server kan vara mer känslig för SQL-injections än exempelvis MySQL och man måste därför vara ytterst försiktig när man skall blanda in data från användare i sina queries. Om vi antar att variabeln `id` i nedanstående kommando kommer från användaren skulle en elak sådan kunna ställa till oreda,

```
SqlCommand command =  
    new SqlCommand("DELETE FROM Products WHERE ProductID = " + id, connection);
```

C# - Fel, query med risk för SQL-injection

Lösningen är att använda i databasvärlden så kallade *named parameters*, eller bara "parametrar" på ren svenska. Enkelt uttryckt innebär det att man ersätter det värde som skall skrivas in i queryn av användaren med en temporär parameter som sedan fylls i efteråt och systemet ser till att citations-tecken och annan "farlig" data blir korrekt behandlade,

```
SqlCommand command =  
    new SqlCommand("DELETE FROM Products WHERE ProductID = @ID", connection);  
command.Parameters.AddWithValue("@ID", id);  
command.Connection.Open();  
command.ExecuteNonQuery();
```

C# - Rätt, systemet ser till att ingen SQL-injection är möjlig

När det gäller vår gamla kamrat `SqlDataAdapter` som vi använt för att fylla `DataSet` med data från `SELECT`-satser har vi hittills skickat in vår SQL-query som ren text som första argument och `connectionsträngen` som andra. Fördelen med den lösningen är, som nämndes på föregående sida, att man slipper definiera en `SqlConnection`, men tyvärr ger den inte något skydd mot SQL-injections,

```
private void SearchProducts(string name)  
{  
    string query = "SELECT * FROM Products WHERE Name LIKE %" + name + "%";  
  
    SqlDataAdapter da = new SqlDataAdapter(query, "Data Source=...");  
    DataSet ds = new DataSet();  
    da.Fill(ds);  
    repeater.DataSource = ds;  
    repeater.DataBind();  
}
```

C# - Fel, query med risk för SQL-injection

Lösningen är att istället för att skicka in SQL-queryn som ren text använda ett `SqlCommand`-objekt, vilket i sin tur för med sig att vi måste ha en `SqlConnection` (vi slipper dock öppna den).

```
private void SearchProducts(string name)  
{  
    using(SqlConnection connection = new SqlConnection("Data Source=..."))  
    {  
        string query = "SELECT * FROM Products WHERE Name LIKE @Name";  
  
        SqlCommand command = new SqlCommand(query, connection);  
        command.Parameters.AddWithValue("@Name", "%" + name + "%");  
  
        SqlDataAdapter da = new SqlDataAdapter(command);  
        DataSet ds = new DataSet();  
        da.Fill(ds);  
        repeater.DataSource = ds;  
        repeater.DataBind();  
    }  
}
```

C# - Rätt, SqlDataAdapter använder ett SqlCommand-objekt istället för en ren text-query

Att läsa från en databas utan databinding

Konceptet med databinding är väldigt användbart när vi skall presentera data, men ibland behöver man ju direkt åtkomst till innehållet i en tabell. För detta behöver vi varken ett DataSet eller en SqlDataAdapter utan istället en ny bekantskap i datatypen SqlDataReader som härstammar från namnrymden System.Data.SqlClient. En SqlDataReader fungerar ungefär så som vi kanske är mest vana vid, det vill säga man öppnar en anslutning till en databas och läser rad efter rad tills man har slut på data. I övrigt är tillvägagångssättet ganska likt från tidigare exempel med ExecuteNonQuery,

- 1 Skapa en koppling mot databasen (SqlConnection)
- 2 Skapa en SQL-query (SqlCommand)
- 3 Öppna databaskopplingen
- 4 Hämta en SqlDataReader baserat på queryn
- 5 Läs av SqlDataReader tills det är slut på rader
- 6 Stäng SqlDataReader
- 7 Stäng databaskopplingen

```
string query = "SELECT TOP 10 ProductID, ProductName FROM Products";
using (SqlConnection connection = new SqlConnection(connectionstring)) // 1
{
    SqlCommand command = new SqlCommand(query, connection); // 2
    connection.Open(); // 3

    SqlDataReader reader = command.ExecuteReader(); // 4

    while (reader.Read()) // 5
    {
        Console.WriteLine("ID = {0}, Name = {1}\n",
            reader["ProductID"],
            reader["ProductName"]);
    }

    reader.Close(); // 6
} // 7
```

C# - Läser 10 produkter från en databas och dumpar ut id och namn i konsolen

Det är ett vanligt scenario att vi bara vill ha ut ett värde från våran tabell, exempelvis från en COUNT som returnerar antal matchande rader på en SQL-query. För detta behöver vi inte använda oss av en SqlDataReader, även om det naturligtvis går, utan SqlCommand har en fiffig metod i ExecuteScalar som helt enkelt returnerar första värdet på den första kolumnen som SQL-queryn har returnerat tillbaka. Följande två exempel gör med andra ord samma sak,

```
SqlCommand command = new SqlCommand("SELECT COUNT(*) FROM Products", connection);
connection.Open();
SqlDataReader reader = command.ExecuteReader();
reader.Read();
int products = Convert.ToInt32(reader[0]);
reader.Close();
```

C# - Skapar en SqlDataReader, läser en rad, plockar ut värdet från första kolumnen

```
SqlCommand command = new SqlCommand("SELECT COUNT(*) FROM Products", connection);
connection.Open();
int products = Convert.ToInt32(command.ExecuteScalar());
```

C# - Plockar också ut värdet från första kolumnen utan att använda en SqlDataReader

ASP.NET Uppgift 12-1

Glöm inte att binda om din repeater efteråt för att uppdatera tabellen på sidan när du har modifierat innehållet i databasen.

Återanvänd databasen med produkter från uppgift 11-3 och lägg till en "ta bort"-knapp till produktlistningen, lämpligen med en bild på en soptunna eller ett kryss. Använd en Imagebutton-kontroll för att skriva ut en bild som gör en PostBack på samma sätt som en Button-kontroll när man klickar på den. När man klickar på "ta bort"-knappen skall produkten plockas bort från databasen.

ASP.NET Uppgift 12-2

Skriv en simpel sökfunktion till dina produkter genom att lägga till en textbox och en knapp högst upp på sidan (tips: du har mer eller mindre all kod du behöver på sidan om SQL-injections).

ASP.NET Uppgift 12-3

Skriv någonstans på sidan ut exakt hur många produkter som finns i databasen (använd COUNT(*)).

ASP.NET Uppgift 12-4

Ovanför sökfunktionen på sidan lägger du in en textbox för produktnamn och en textbox för produktpris, samt en knapp för att lägga till produkter. När man klickar på knappen skall, föga förvånande, produkten läggas till i databasen och listningen av produkter uppdateras.

ASP.NET Uppgift 12-5 (Frivillig)

Uppdatera uppgift 12-3 så att den listar både hur många produkter som finns i databasen och antal produkter som för tillfället visas på sidan ("Visar x produkter av y antal"). Försök klura ut hur du kan göra detta utan att behöva göra ytterligare en COUNT-query mot databasen.

ASP.NET Uppgift 12-6 (Frivillig)

Lägg till någon funktion på sidan som sorterar produkterna baserat på stigande/fallande namn eller pris. Kom ihåg att behålla sorteringen även när man söker på produktnamn.

Att centralisera anropen till databasen

Att skriva en större databasdriven applikation för ofta med sig effekten att en stor del av källkoden hanterar anslutningar och läsningar till och från databasen, vilket blandat med logiken för själva applikationen ofta gör källkoden svåröverskådligt och i förlängningen mer komplicerad att underhålla och felsöka i.

Idag finns det ett antal olika lösningar för att underlätta kommunikationen till databasen, några svär troget vid sina OR/M-system (som tar bort all SQL) och andra anser att så mycket kod som möjligt skall skrivas i *Stored Procedures* för att på så vis helt flytta alla SQL-kommandon från applikationen direkt in i databasen istället. Båda systemen har sina för- och nackdelar, ett OR/M känns ofta begränsande vad gäller mer avancerade databasfrågor och att behöva skriva en ny Stored Procedure för varje liten SQL-sats man vill utföra blir i längden både tidskrävande och minst lika svårt att underhålla.

En tredje variant, vilket vi kommer använda här, är att istället skapa en eller, om så behövs, flera separata klasser med statiska metoder där all kommunikation med databasen förs. Klassen får gärna heta något så tråkigt som Database och innehålla beskrivande metodnamn som AuthenticateUser, CreateNewPost, GetPostById, och så vidare. Exempel,

```
public class Database
{
    public static string ConenctionString
    {
        get {
            return ConfigurationManager.ConnectionStrings["DB"].ConnectionString;
        }
    }

    public static void CreateNewPost(string subject, string post)
    {
        using (SqlConnection connection = new SqlConnection(ConenctionString))
        {
            SqlCommand command = new SqlCommand(
                "INSERT INTO Posts(Subject, Post) VALUES(@subject, @post)",
                connection);
            command.Parameters.AddWithValue("@subject", subject);
            command.Parameters.AddWithValue("@post", post);
            connection.Open();
            command.ExecuteNonQuery();
        }
    }
}
```

C# - Database.cs. Eftersom det är så mycket att behöva skriva för att få ut connection-strängen, vilken kommer behövas i varje metod, så har vi kapslat in den i en lättare att använda publik egenskap högst upp i klassen.

Ponera nu en händelsehanterare på en sida som haterar en knapptryckning från ett formulär med två textboxar för att skapa en ny post i databasen. Istället för att behöva dra in någon databaskod alls på den sidan räcker det nu med att man skriver, exempelvis,

```
public void SubmitPost(object sender, EventArgs e)
{
    Database.CreateNewPost(subject.Text, post.Text);
}
```

C#

ASP.NET Uppgift 13-1

Skriv ett fungerande och databasdrivet loginsystem med layout efter tycke och smak. Skapa en ny databas eller återanvänd en gammal. Skapa en tabell för användare kallad "Users". Tabellen skall innehålla minst följande fält för den här uppgiften,

- Ett automatiskt uppdaterat fält för ID
- Användarnamn
- Lösenord
- Tidsangivelse när användaren skapades (DateTime-kolumn med default getDate() i SQL-Server)

Är du haj på SQL kan du lägga till ett unikt index på kolumnen med användarnamn för att öka integriteten i databasen.

Skriv en sida för skapandet av nya konton, sidan skall bestå av en textbox för användarnamn och en för lösenord. Kontrollera så att lösenordet inte är tomt och att användarnamnet inte redan finns i databasen. När allt validerar skall en INSERT göras och databasen uppdateras med en ny användare.

Skriv en ny sida med ett loginformulär som kontrollerar användarnamn och lösenord mot databasen. Lyckas användaren logga in skall han eller hon bli inloggad och hamna i en skyddad katalog som inte annars går att komma åt. Se avsnittet om autentisering och uppgift 9-1 om det står still i huvudet.

Skapa en separat klass efter exemplet nedan som hanterar alla databasanrop,

```
public class Database
{
    public static string ConenctionString
    {
        get {
            return ConfigurationManager.ConnectionStrings["DB"].ConnectionString;
        }
    }

    public static void AddUser(string username, string password)
    {
        // Lägger till en ny användare i databasen. En förbättrad version skulle
        // istället kunna returnera en int med den nya användarens ID.
    }

    public static bool Authenticate(string username, string password)
    {
        // Returnerar true om username + password matchar en rad i databasen.
    }

    public static bool UserExists(string username)
    {
        // Returnerar true om en username matchar en rad i databasen.
    }
}
```

C#

Paging genom PagedDataSource

När en datakälla innehåller mycket data är det ofta nödvändigt att dela upp utskriften till användaren på flera sidor och ha något typ av navigationssystem för att hoppa mellan dem, vanligen kallat *paging* på engelska. När datakällan kommer från en databas är det *ibland* möjligt att kunna utföra en sådan selektion direkt i databasen - det vill säga "returnera alla rader mellan rad x och y". Typiskt nog är SQL Server, den absolut vanligaste databasen i ASP.NET-sammanhang, en av de databaserna där man inte har någon inbyggd konstruktion för detta. En datakälla kan som tidigare avhandlats också komma från andra saker än en databas, exempelvis en array, vilket har tvingat fram klassen PagedDataSource; en generell lösning för att stega igenom all typ av data som man kan binda mot.

PagedDataSource är en relativt enkel klass att använda och kan ses ungefär som ett lager som ligger mellan själva kontrollen och datakällan och ser till att rätt data visas för användaren. För att ha ett exempel att utgå ifrån kan vi använda en väldigt simpel Repeater-kontroll som använder sig av en array innehållandes namnen på olika programspråk. Under utskriften finns två knappar som skall användas för att bläddra mellan programspråken så att endast fem stycken visas samtidigt på varje sida. Bredvid knapparna finns en label som kommer tala om hur många sidor som finns totalt att bläddra igenom,

```
<b>Programspråk</b><br/>
<asp:Repeater runat="server" ID="LanguageRepeater">
  <ItemTemplate>
    <%# Container.DataItem %>
  </ItemTemplate>
</asp:Repeater>
<hr/>
<asp:Button runat="server" ID="PrevButton" text="Prev"/>
<asp:Button runat="server" ID="NextButton" text="Next"/>
Sidor: <asp:Label runat="server" id="Pages" />
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
        BindRepeater();
}

private void BindRepeater()
{
    string[] languages = {
        "C#", "Perl", "Java", "Pascal", "PHP", "Ruby", "Python", "C++",
        "Lisp", "Delphi", "C", "Fortran", "Basic", "Smalltalk", "Rexx"
    };

    LanguageRepeater.DataSource = languages;
    LanguageRepeater.DataBind();
}
```

C#

Programspråk

C# Perl Java Pascal PHP Ruby Python C++ Lisp Delphi C Fortran Basic Smalltalk Rexx

Prev

Next

Sidor:

Som exemplet visar har vi inte använt oss av PagedDataSource än utan skriver tills vidare ut alla programspråken i en följd. De två knapparna har inte några händelsehanterare för att navigera och Label-kontrollen förblir tom.

För att kunna "dela upp" datakällan i flera delar, så att användaren bara ser fem programspråk i taget med andra ord, måste vi introducera PagedDataSource mellan vår repeater och datakällan. De viktigaste egenskaperna på en instans av PagedDataSource som man nästan alltid använder sig av är,

DataSource

Naturligtvis, vilken datakälla som skall användas.

AllowPaging

Måste man alltid sätta till **true** för att PagedDataSource skall fungera över huvudtaget (varför den inte är **true** som standard har författaren till den här texten inte en aning om).

PageSize

Hur många iterationer (delar av datakällan) som skall visas åt gången - i det här fallet fem stycken.

CurrentPageIndex

Vilken "sida" som skall visas, det vill säga sidan noll genererar datan mellan 1-5, sidan ett mellan 6-10, och så vidare, helt beroende på vad PageSize är satt till.

PageCount

Det totala antal sidor som finns att visa.

För att slå ihop detta i någon meningsfull kontext kan vi modifiera BindRepeater-metoden från föregående sida enligt följande,

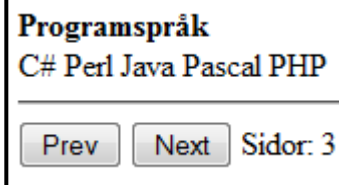
```
protected void BindRepeater()
{
    string[] languages = {
        "C#", "Perl", "Java", "Pascal", "PHP", "Ruby", "Python", "C++",
        "Lisp", "Delphi", "C", "Fortran", "Basic", "Smalltalk", "Rexx"
    };

    PagedDataSource pager = new PagedDataSource();
    pager.AllowPaging = true;
    pager.PageSize = 5;
    pager.CurrentPageIndex = 0;
    pager.DataSource = languages;

    LanguageRepeater.DataSource = pager;
    LanguageRepeater.DataBind();

    Pages.Text = Convert.ToString(pager.PageCount);
}
```

C#



Nu visas bara de fem första programspråken, och skulle vi ändra CurrentPageIndex till 1 i C#-koden skulle nästa fem programspråk visas. Vidare kan vi med PageCount skriva ut hur många sidor som det totalt finns att visa men knapparna för att kunna navigera saknar fortfarande händelsehanterare.

Ett problem som uppstår är att vi nu måste börja hålla reda på vilken sida som användaren befinner sig på så att vi kan gå framåt och bakåt utifrån den. Det finns två sätt att lösa det på, antingen använder vi en variabel i query-strängen (tänk "?page=1", se kapitlet om avancerad paging för ett exempel), eller - vilket är enklare men kanske inte alltid bättre - så introducerar vi en egenskap på sidan som kapslar in ett heltal för aktuell sida i ViewState,

```
protected int PagerIndex
{
    get { return Convert.ToInt32(ViewState["PagerIndex"]); }
    set { ViewState["PagerIndex"] = value; }
}
```

C# - Det kan vara värt att notera att `Convert.ToInt32(null)` returnerar heltalet noll, vilket gör att `PagerIndex` utan något tidigare tilldelat värde kommer returnera 0, eller första sidan med andra ord.

Vi kan nu använda `PagerIndex` för att hålla reda vilken sida användaren befinner sig på. Istället för att alltid visa första sidan uppdaterar vi `BindRepeater`-metoden till att visa den sidan som `PagerIndex` anger,

```
pager.CurrentPageIndex = PagerIndex;
```

C#

Händelsehanterarna för navigationsknapparna behöver nu bara öka respektive minska `PagerIndex` och sedan anropa `BindRepeater()` igen för att avancera fram och tillbaka,

```
...
<asp:Button runat="server"
             ID="PrevButton" text="Prev" OnCommand="MoveIndex" CommandName="Prev"/>
<asp:Button runat="server"
             ID="NextButton" text="Next" OnCommand="MoveIndex" CommandName="Next"/>
...
```

ASP.NET

```
protected void MoveIndex(object sender, CommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "Prev":
            PagerIndex--;
            break;
        case "Next":
            PagerIndex++;
            break;
    }
    BindRepeater();
}
```

C#

Skulle vi provköra exempelkoden nu skulle det med hjälp av navigationsknapparna gå att hoppa fram och tillbaka mellan de olika programspråken. Det finns dock inget som hindrar användaren från att klicka sig fram eller bakåt utanför vad som borde vara tillåtet, vilket resulterar i att om användaren är på första sidan och klickar på "Prev" kommer `PagerIndex` bli satt till -1 och sidan kraschar.

Den mest uppenbara lösningen på det problemet är att introducera en **if**-sats som kontrollerar om `PagerIndex` är lika med eller över noll och under antal sidor (`pager.PageCount`) och om inte sätta `PagerIndex` till första sidan igen. Det fungerar, men en ännu snyggare lösning är att se till så att när användaren är på första eller sista sidan går inte Prev- respektive Next-knappen att klicka på över huvudtaget. Även om det naturligtvis är möjligt att själv hålla reda på om användaren är på första eller sista sidan (`PagerIndex` är noll eller `PagerIndex` är lika med `pager.PageCount - 1`) finns det i `PagedDataSource` två smidiga booleska egenskaper `IsFirstPage` och `IsLastPage` för detta. Sammanslaget blir den kompletta C#-koden följande,

```
protected int PagerIndex
{
    get { return Convert.ToInt32(ViewState["PagerIndex"]); }
    set { ViewState["PagerIndex"] = value; }
}

protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
        BindRepeater();
}

protected void BindRepeater()
{
    string[] languages = {
        "C#", "Perl", "Java", "Pascal", "PHP", "Ruby", "Python", "C++",
        "Lisp", "Delphi", "C", "Fortran", "Basic", "Smalltalk", "Rexx"
    };

    PagedDataSource pager = new PagedDataSource();
    pager.AllowPaging = true;
    pager.PageSize = 5;
    pager.CurrentPageIndex = PagerIndex;
    pager.DataSource = languages;

    LanguageRepeater.DataSource = pager;
    LanguageRepeater.DataBind();

    Pages.Text = Convert.ToString(pager.PageCount);

    if (pager.IsFirstPage)
        PrevButton.Enabled = false;
    else
        PrevButton.Enabled = true;

    if (pager.IsLastPage)
        NextButton.Enabled = false;
    else
        NextButton.Enabled = true;
}

protected void MoveIndex(object sender, CommandEventArgs e)
{
    switch (e.CommandName)
    {
        case "Prev":
            PagerIndex--;
            break;
        case "Next":
            PagerIndex++;
            break;
    }

    BindRepeater();
}
}
```

C#

Programspråk
C# Perl Java Pascal PHP

Prev Next Sidor: 3

Paging med data från en databas

Varför exemplet i föregående kapitel använde sig av en array som datakälla var för att förenkla det, i ett kanske mer realistiskt exempel skulle datakällan exempelvis kunna komma från en `SqlDataAdapter` med ett `DataSet` innehållande data från en `SELECT`-sats från en databas. Tyvärr kan inte en `PagedDataSource` direkt använda ett `DataSet` som datakälla, då ett `DataSet` teoretiskt kan bestå av flera tabeller, istället måste vi gå lite djupare ner och peka på exakt vilken tabelldata i datasettet som skall användas. För att binda datan från "första tabellen" i ett `DataSet` till en `PagedDataSource` och vidare till en `Repeater`-kontroll använder man egenskapen `Tables[0].DefaultView` på `DataSet` enligt följande exempel,

```
public void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
        BindGuestbook(0);
}

// Returnerar ett dataset med gästboksdata
private DataSet GetGuestbook()
{
    DataSet ds = new DataSet();

    using (SqlConnection connection = new SqlConnection("Data Source=..."))
    {
        SqlCommand cmd = new SqlCommand(@"SELECT PostID, PostText FROM Guestbook",
                                         connection);
        SqlDataAdapter adapter = new SqlDataAdapter(cmd);
        adapter.Fill(ds);
        return ds;
    }
}

// Binder datan till en repeater och bläddrar till en specifik sida
private void BindGuestbook(int pageToView)
{
    DataSet ds = GetGuestbook();

    PagedDataSource pager = new PagedDataSource();
    pager.AllowPaging = true;
    pager.PageSize = 15;
    pager.DataSource = ds.Tables[0].DefaultView;
    pager.CurrentPageIndex = pageToView;

    GuestbookRepeater.DataSource = pager;
    GuestbookRepeater.DataBind();
}
```

- Ett alternativ till exemplet ovan som man ibland dyker på när man bara använder en tabell är att istället för ett `DataSet` använda datatypen `DataTable` (vilket är vad `Table[0]` i `DataSet` är). Metoden `adapter.Fill()` fungerar likadant på båda .

```
<asp:Repeater runat="server" ID="GuestbookRepeater">
  <ItemTemplate>
    <b><# DataBinder.Eval(Container.DataItem, "PostID") %></b><br/>
    <# DataBinder.Eval(Container.DataItem, "PostText") %>
  </ItemTemplate>
</asp:Repeater>
```

ASP.NET

Exemplet i sig implementerar ingen form av paging men kan i framtiden hoppa till en ny sida genom att vi anropar metoden `BindGuestbook` med ett heltal större än noll.

ASP.NET Uppgift 14-1

Arbeta vidare med sidan du förhoppningsvis gjorde i uppgift 12. Lägg till ytterligare produkter i databasen så att det finns i alla fall en 20 stycken (har du ingen fantasi kan du bara kalla dem "Produkt A", "Produkt B", och så vidare).

Ändra så att bara fem produkter visas per sida och med navigationsknappar längst ner för att bläddra mellan sidorna. Bläddringsfunktionen skall också fungera vid sökningar av produkter.

ASP.NET Uppgift 14-2

Bryt ut all databaskod från sidan och lägg det i en egen fil. Kom ihåg att filen skall vara generell så att du (teoretiskt) kan använda den från andra sidor. Det innebär att du inte bör skicka kontroller från din sida som parametrar in i metoderna, låt istället metoderna returnera data och bind upp kontrollerna på den aktuella sidan. Skriv alltså något i stil med,

```
public PagedDataSource SearchProducts(string text, int pageToView)
{
    PagedDataSource pager = new PagedDataSource();
    // databaskod för att söka och hoppa till rätt sida
    return pager
}
```

C# - Database.cs, returnerar ett PagedDataSource med data som kan bindas från alla sidor som behöver söka efter produkter.

ASP.NET Uppgift 14-3 (Frivillig)

Förutsatt att du har gjort uppgift 12-4, ändra händelsehanteraren för att lägga till produkter så att sidan uppdateras utan att nollställas. Med andra ord, har man gjort en sökning på "test" och fått sju produkter, hoppat till sidan två och väl där beslutat sig för att lägga till en ny produkt som matchar "test" så skall man bli kvar på sidan två och produkten dyka upp (förutsatt att den hamnar sist i databasen alltså, hamnar den bland de fem första kommer den sista produkten från förstasidan istället ramla in på sidan två).

Avancerad Paging

En annan relativt vanlig lösning på det här problemet (men i författarens tycke fruktansvärt ful och förkastlig), är att bygga upp en textsträng med HTML-länkar för hand i C#-koden och sedan spotta ut den i exempelvis en label.

Något förvånande får man väldigt lite hjälp från ASP.NET när man vill implementera en mer avancerad typ av paging, som exempelvis länkar med siffror för varje sida som användaren kan klicka på (typ som Google gör). En fördel med att använda "vanliga" länkar är att de till skillnad mot kontroller som postar tillbaka data går att länka till och är lättare att indexera för sökmotorer, något som på senare år i vissa fall har blivit mer prioriterat i branschen än att faktiskt ha något vettigt innehåll bakom.

För att snabbt få lite exempelkod kan vi återanvända och bygga vidare på den enkla gästboken från föregående kapitel. Så som den är skriven nu visas alltid bara de 15 första inläggen. Uppgiften blir att implementera "sökoptimerade" URL:er för navigation ovanför gästboken som kan användas för att hoppa mellan de olika sidorna, 15 inlägg i taget.

För detta skriver vi först en ny Repeater ovanför gästboken som innehåller layouten för länkarna,

```
Sida [  
  <asp:Repeater runat="server" ID="Pager">  
    <ItemTemplate>  
      <a href="default.aspx?page=<## Container.DataItem %>">  
        <## Container.DataItem %>  
      </a>  
    </ItemTemplate>  
  </asp:Repeater>  
</Sida >
```

ASP.NET

I C#-koden måste det finnas något som binder datan till vår nya Pager-repeater så att den genererar rätt antal länkar. För detta kommer vi skapa en "dummy"-array med lika många siffror som antal sidor som skall visas. Vi kommer göra den enklaste varianten, att helt enkelt visa klickbara länkar som börjar på första sidan och slutar med den sista. Det här fungerar prima så länge som det inte finns för mycket data (att skriva ut länkar för exempelvis 100 sidor ser inte snyggt ut). Vår array kommer med andra ord börja på 1 och sluta med totalt antal inlägg i gästboken dividerat med antal inlägg per sida som skall visas. Alla som har skrivit en egen paging-funktion i ett annat språk borde känna igen tankegången.

För detta behöver vi först en simpel metod som returnerar totalt antal inlägg,

```
private int GetGuestbookCount()  
{  
  using (SqlConnection connection = new SqlConnection("Data Source=..."))  
  {  
    SqlCommand command = new SqlCommand(@"SELECT COUNT(*) FROM Guestbook",  
                                          connection);  
    connection.Open();  
    return Convert.ToInt32(command.ExecuteScalar());  
  }  
}
```

C#

Därefter kan vi skriva metoden `BindPager` som binder våra `Repeater`. Eftersom vi dividerar det totala antalet inlägg med antal per sida är det viktigt att vi rundar av uppåt, annars finns risken att vi får exempelvis 7,2 sidor som avrundat neråt blir 7 - vilket gör att vi missar några av de sista inläggen som skulle finnas på sidan 8. För att alltid avrunda uppåt finns den väldigt vanligt förekommande matematiska *Ceil*-funktionen, i .NET implementerad som `Math.Ceiling`,

```
private void BindPager()
{
    // Totalt antal sidor vi kan visa, avrundat uppåt
    int total = (int)Math.Ceiling((double)GetGuestbookCount() / 15);

    // Skapa en ny dummy-array med lika stor storlek som antal sidor
    int[] dummy = new int[total];

    // Fyll den med siffror (1, 2, 3, 4, osv...)
    for (int i = 0; i < total; i++)
    {
        dummy[i] = i + 1;
    }

    Pager.DataSource = dummy;
    Pager.DataBind();
}
```

C#

```
Sida: [
  <a href="default.aspx?page=1">1</a>
  <a href="default.aspx?page=2">2</a>
  <a href="default.aspx?page=3">3</a>
  <a href="default.aspx?page=4">4</a>
]
```

Sida: [[1](#) [2](#) [3](#) [4](#)]

Resultat

Vad som saknas nu är att faktiskt byta till rätt sida när användaren klickar på en länk. För detta måste vi fånga upp query-strängen i `Page_Load`, något som avhandlas i slutet av häftet under avsnittet *Request och Response*, men som borde vara relativt enkelt att förstå ändå,

```
public void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
    {
        BindPager();

        // Har inte användaren klickat på en länk visas första sidan
        if (Request["page"] == null)
            BindGuestbook(0);
        else
            BindGuestbook(Convert.ToInt32(Request["page"]) - 1);
    }
}
```

C#

Även om det här fungerar saknas det en del för att den här koden skulle få gå i produktion. Exempelvis finns det inget som hindrar att användaren för hand ändrar "page"-variabeln i sin webbläsare till ett värde utanför det tillåtna, och naturligtvis borde den aktuella sidan som är vald inte vara en klickbar länk utan istället på något sätt bli markerad (En `OnItemDataBound` på `Pager` och kontrollera om aktuell länksiffra stämmer med vald sida är en bra början).

ASP.NET Uppgift 15-1

Arbeta vidare med sidan från uppgift 14. Mellan knapparna som navigerar fram och tillbaka lägger du in navigationslänkar i form av siffror. Det här är kanske inte så enkelt som det låter, till skillnad mot att klicka på navigationsknapparna så är det här vanliga HTML-länkar vilket gör att sidan inte utför en PostBack längre. Du kan därför inte spara information i ViewState utan måste exempelvis "få med dig" ett sökresultat i query-strängen (tänk typ "?search=test&page=2").

ASP.NET Uppgift 15-2

Lägg till kod som fetmarkerad den valda sidan i sifferlänkarna och ser till så att den inte går att klicka på.

ASP.NET Uppgift 15-3

Lägg till kod som gör att det inte går att manipulera URL:en i webbläsaren för hand för att krascha sidan, det vill säga se till så att variabeln i query-strängen både innehåller ett heltal och är större än 0 och mindre eller lika med antal sidor som kan visas.

ASP.NET Uppgift 15-4 (Frivillig)

Förhoppningsvis har du en ganska robust paging-funktion för dina produkter nu, men vad händer om du lägger till 100 produkter till? Det blir ganska många sifferlänkar att skriva ut och ser inget vidare ut. Skriv en lösning som begränsar antalet utskrivna länkar men fortfarande gör det möjligt att navigera runt. Står det helt still i huvudet, kolla hur exempelvis Google eller något större forum gör.

ASP.NET AJAX

När ASP.NET togs fram hade den funktionaliteten som "AJAX" bidrar med inte ens något enhetligt namn (benämningen *Asynchronous JavaScript And XML* "uppfanns" först 2005), det närmaste var "DHTML" som vid den tidpunkten mest var en enda röra av JavaScript och CSS-hack. Med andra ord var möjligheten att asynkront jobba med data på det sättet som många dynamiska hemsidor idag är byggda runt inte något som ens fanns på kartan när ASP.NET 1.0 utvecklades, och detta gjorde att när JavaScript fick sin renässans halvvägs in på 2000-talet var det allt annat än självklart hur man skulle, om det ens var möjligt, få AJAX att integrera med ASP.NETs modell med PostBacks, där hela sidan postas tillbaka i ett stycke för att sedan renderas om på nytt.

Samtidigt som Microsoft så smått började utveckla AJAX-funktionalitet för ASP.NET i en beta kallad *Atlas* dök det upp alternativa AJAX-ramverk för ASP.NET, dock inga som direkt slog igenom på bred front. Efter en lång utvecklingstid släpptes till slut Atlas i en stabil version under det föga upphetsande namnet *ASP.NET AJAX*. I och med ASP.NET 3.5 distribueras ASP.NET AJAX nu mera tillsammans med ASP.NET, i tidigare versioner av ramverket måste det laddas ner och installeras som en separat modul, något som inte är helt enkelt.

Med ASP.NET AJAX lyckades Microsoft med konststycket att både integrera AJAX-funktionalitet i ASP.NET samtidigt som befintliga ASP.NET-sidor inte behövdes skrivas om i någon större utsträckning för att utnyttja det. ASP.NET AJAX fungerar som ett lager ovanpå den redan beprövade PostBack-modellen - man kan med andra ord först utveckla en ASP.NET-sida "som vanligt" för att sedan, när sidan är klar, enkelt lägga på AJAX-funktionalitet i efterhand utan att behöva ändra en enda rad C#-kod.

ASP.NET AJAX Control Toolkit

Ungefär samtidigt som ASP.NET AJAX kom ut på marknaden släppte Microsoft *ASP.NET AJAX Control Toolkit*, vilket *inte* är samma sak som ASP.NET AJAX, utan är en uppsättning mer dynamiska ASP.NET-kontroller som använder JavaScript för att åstadkomma effekter som animationer och liknande. ASP.NET AJAX Control Toolkit är inte en del av ASP.NET AJAX och följer inte heller med ASP.NET 3.5 utan måste fortfarande laddas ner separat.

Intresset för ASP.NET AJAX Control Toolkit verkar ha svalnat avsevärt från Microsofts sida på senare år då projektet flyttats över till *Codeplex*, Microsofts hemsida för öppen källkod, och släpptes under en fri licens utan någon officiell support.

Aktivera ASP.NET AJAX med ScriptManager

För att kunna använda sig av ASP.NET AJAX måste en sida först inkludera en speciell kontroll, ScriptManager. Den här kontrollen placerar precis som alla andra kontroller inuti form-taggen och ser till att Microsofts JavaScript-bibliotek som ASP.NET AJAX använder sig av inkluderas när sidan renderas. Då ScriptManager måste förekomma på sidan före alla kontroller som använder sig av ASP.NET AJAX är det vanligt att man helt enkelt inkluderar den direkt efter form-taggen.

Tyvärr kan det fortfarande än idag behövas lite handpåläggning innan ASP.NET AJAX faktiskt fungerar med en vanliga ASP.NET-sida, detta manifesterar sig i så fall i att ScriptManager inte finns i IntelliSense-listan över tillgängliga kontroller. Skulle så vara fallet måste följande aningen kryptiska sektion läggas till i applikationens web.config under sektionen <system.web>.

```
<pages>
  <controls>
    <add tagPrefix="asp"
        namespace="System.Web.UI"
        assembly="System.Web.Extensions,
                Version           = 1.0.61025.0,
                Culture           = neutral,
                PublicKeyToken    = 31bf3856ad364e35" />
  </controls>
</pages>
```

web.config - Det går nog snabbare att söka på Google och kopiera därifrån än att skriva av.

UpdatePanel

I ASP.NET AJAX har kontrollen updatePanel en central roll och används för att på sidan omsluta en eller flera andra "vanliga kontroller" och markera att dessa skall kunna uppdateras individuellt från resten av sidan, alltså i enklare termer utan att hela sidan behöver laddas om - något som Microsoft kallar för *Partial-Page Rendering*. För att förstå hur detta ter sig kan vi utgå från nedanstående exempel som visar en simpel ASP.NET-sida utan någon AJAX-funktionalitet; när man klickar på "Uppdatera" laddas hela sidan om och kontrollen "Timer2" uppdateras med en ny tid.

```
<form runat="server">
  Timer 1: <asp:Label runat="server" id="Timer1"/><br />
  Timer 2: <asp:Label runat="server" id="Timer2"/><br />
  <asp:LinkButton onclick="UpdateTimer2" runat="server" Text="Uppdatera"/>
</form>
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
    {
        Timer1.Text = DateTime.Now.ToString();
        Timer2.Text = DateTime.Now.ToString();
    }
}

protected void UpdateTimer2(object sender, EventArgs e)
{
    Timer2.Text = DateTime.Now.ToString();
}
```

C#

```
Timer 1: 2009-09-12 19:05:43
Timer 2: 2009-09-12 19:05:48
Uppdatera
```

Om vi nu i efterhand vill lägga till ASP.NET AJAX och få "Timer2" att uppdatera sig utan att behöva ladda om hela sidan behöver vi bara lägga till en ScriptManager högst upp och omsluta området vi vill uppdatera samt PostBack-länken med en UpdatePanel,

```
<form runat="server">
  <asp:ScriptManager runat="server" />
  Timer 1: <asp:Label runat="server" id="Timer1"/><br />
  <asp:UpdatePanel runat="server">
    <ContentTemplate>
      Timer 2:
      <asp:Label runat="server" id="Timer2"/><br />
      <asp:LinkButton onclick="UpdateTimer2" runat="server" Text="Uppdatera"/>
    </ContentTemplate>
  </asp:UpdatePanel>
</form>
```

ASP.NET

Sidan ser identisk ut och C#-koden är oförändrad, men istället för att hela sidan laddas om när användaren klickar på länken så sker det i bakgrunden ("asynkront") och "Timer2" uppdaterar sig utan att webbläsaren blinkar till, så som det ofta ser ut när det görs en vanlig PostBack. Det är viktigt att både kontrollen som postar tillbaka data och de delarna som skall uppdateras ligger i en UpdatePanel. Skulle vi exempelvis försöka uppdatera "Timer1" istället händer till synes ingenting förrän en kontroll utanför en UpdatePanel gör en "vanlig" PostBack och hela sidan renderar om sig.

En UpdatePanel renderar sig som en <div> runt kontrollerna i webbläsaren vilket inte alltid är så bra layoutmässigt. Vill man istället rendera en UpdatePanel som en för layouten ofta mindre påverkande går det att lägga till attributet RenderMode="Inline".

Även om ovanstående exempel bara uppdaterar en textsträng är UpdatePanel inte begränsad till så enkla operationer, vi skulle teoretiskt kunna lägga hela sidor inuti en enda stor UpdatePanel och få "AJAX"-funktionalitet överallt på några sekunder. För den här smidigheten finns det dock ett pris att betala:

En av de stora fördelarna som ofta associeras med att använda "AJAX" är att det är relativt lite data som flyttas mellan klient och server. Istället för att posta ett helt formulär och sedan rendera om hela sidan kan utvecklaren välja att skicka en liten del från klienten och sedan svara tillbaka med bara den lilla biten som skall uppdateras. Det här är inte sant när man använder en UpdatePanel - för att ASP.NET skall kunna fungera "som vanligt" när det sker en asynkron PostBack måste inte bara alla kontroller inuti den (eller de) panelerna som skall uppdateras skickas till servern, även den ofta flera kilobyte stora ViewState-strängen måste följa med. ASP.NET svarar sedan tillbaka med en ny ViewState-sträng och färdigrenderade HTML-block som via JavaScript placeras in på rätt ställen på sidan. Allt som allt kan det med andra ord trots "AJAX-funktionaliteten" bli relativt mycket data som flyttas via klient och servern med ASP.NET AJAX jämfört med andra mer lättviktiga ramverk.

I ett senare kapitel kommer vi se hur flera UpdatePanels kan användas för att få ner storleken på nätverkstrafiken.

ASP.NET Uppgift 16-1

Lägg in hela Uppgift 15 i en enda stor `UpdatePanel` så att de delarna som använder sig `PostBacks` arbetar asynkront, och glöm inte att lägga till en `ScriptManager` högst upp på sidan. Får du ingen *Intellisense* på `ScriptManager` innebär det troligen att du måste aktivera ASP.NET genom att göra ändringar i applikationens `web.config`, så som det står beskrivet i avsnittet *Aktivera ASP.NET AJAX med ScriptManager*.

ASP.NET Uppgift 16-2

Skapa en ny sida som listar produkterna från databasen på samma sätt som i uppgift 15. Implementera en paging-funktion med siffror genom en `Repeater`, men använd `LinkButton` istället för "vanliga" länkar. En `LinkButton` fungerar på samma sätt som en vanlig knapp och har exempelvis `onCommand` och `CommandName`-attribut som kan användas för att hålla ett värde när man gör en `PostBack` med dem.

Du kan återvinna det mesta av koden från Uppgift 15 och använda dig av SQL-anropen från din databas-klass.

ASP.NET Uppgift 16-3

Lägg alltihop i en `UpdatePanel` så att pagingen och uppdateringen blir asynkron.

ASP.NET Uppgift 16-4 (Frivillig)

Se till att när man klickar på en av länkarna skall den på något sätt bli markerad och inte gå att klicka på igen.

ASP.NET Uppgift 16-5 (Frivillig)

Komplettera sidan genom att lägga till sökfunktionen från uppgift 15. Naturligtvis skall den också fungera asynkront. För att hålla reda på söktermen får du använda dig av `ViewState` istället för att skicka den via query-strängen i webläsaren.

UpdatePanel med Triggers

Även om kontrollen som skall posta tillbaka data ofta ligger i anslutning till den delen av sidan som skall uppdateras är det naturligtvis inte alltid så. Istället för att då kapsla in stora delar av sidan i en eller flera paneler kan man registrera *Triggers*; kontroller som skall fungera precis som vore de en del av en `updatePanel` trots att de ligger utanför,

```
<form runat="server">
  <asp:ScriptManager runat="server" />
  <asp:UpdatePanel runat="server">
    <ContentTemplate>
      <asp:TextBox runat="server" id="TextBox" />
    </ContentTemplate>
    <Triggers>
      <asp:AsyncPostBackTrigger ControlID="DropDown" />
    </Triggers>
  </asp:UpdatePanel>
</form>

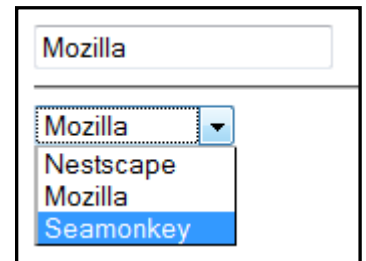
<asp:DropDownList id="DropDown" AutoPostBack="true" runat="server"
  OnSelectedIndexChanged="DropDownChanged">
  <asp:ListItem>Netscape</asp:ListItem>
  <asp:ListItem>Mozilla</asp:ListItem>
  <asp:ListItem>Seamonkey</asp:ListItem>
</asp:DropDownList>
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
        TextBox.Text = DropDownList.SelectedValue;
}

protected void DropDownList_SelectedIndexChanged(object sender, EventArgs e)
{
    TextBox.Text = DropDownList.SelectedValue;
}
```

C#



Att dela upp innehållet i flera UpdatePanels

Anledningen till varför man överhuvudtaget vill använda flera än en `updatePanel` på en sida är som tidigare nämnts för att en stor `updatePanel` skickar relativt mycket data över nätverket; desto färre kontroller som en `updatePanel` måste hantera desto snabbare och effektivare fungerar sidan.

När en sida använder sig av fler än en `updatePanel` är standardförfarandet att om en `updatePanel` uppdaterar sig, så uppdateras också alla andra samtidigt. Detta är visserligen smidigt och löjligt enkelt att jobba med, men inte så effektivt när man bara vill uppdatera en liten del av sidan. För att stänga av den automatiska uppdateringen och därmed få mer kontroll över när en viss `updatePanel` skall uppdatera sig kan vi lägga till attributet `updateMode="Conditional"` (standard är "Always").

Nedanstående exempel låter användaren bygga en levande död armé från tre olika bilder. För att bläddra mellan de tre bilderna används en `MultiView` i en `UpdatePanel` och två navigationsknappar, "<" och ">". Mellan knapparna finns en "OK"-knapp som används för att välja en bild till kollektionen som visas i en annan `UpdatePanel` under. Efter kollektionen av bilder renderas en "Ångra"-knapp som plockar bort senast valda bilden.

Eftersom navigationsknapparna ligger utanför en `UpdatePanel` måste vi lägga till Triggers som talar om att de skall fungera asynkront. Dessa Triggers gör att "<" och ">" bara uppdaterar den översta panelen och "OK"-knappen bara uppdaterar den undre. För att inte panelerna automatiskt skall uppdatera sig så fort en asynkron postback sker måste vi sätta `UpdateMode="Conditional"` på båda.

Eftersom knappen "Ångra" ligger inuti den panelen som den skall uppdatera behöver vi inte ange den som en Trigger. Vill man undvika att en `UpdatePanel` automatiskt uppdaterar sig så fort en kontroll i den gör en asynkron postback används det beskrivande attributet `ChildrenAsTriggers="false"`.

```
<form runat="server">
<asp:ScriptManager runat="server" />
<b>Välj 5 ondingar : </b>
<asp:UpdatePanel runat="server" RenderMode="Inline" UpdateMode="Conditional">
  <ContentTemplate>

    <asp:multiview runat="server" ActiveViewIndex="0" id="viewer">
      <asp:View runat="server"></asp:View>
      <asp:View runat="server"></asp:View>
      <asp:View runat="server"></asp:View>
    </asp:multiview>

  </ContentTemplate>

  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="prv" />
    <asp:AsyncPostBackTrigger ControlID="nxt" />
  </Triggers>
</asp:UpdatePanel>

<asp:Button Text="<" runat="server" id="prv" oncommand="Nav" CommandName="Prev"/>
<asp:Button Text="OK" runat="server" id="add" onclick="Add" />
<asp:Button Text=">" runat="server" id="nxt" oncommand="Nav" CommandName="Next"/>

<br/><b>Din armé:</b><br/>
<asp:UpdatePanel runat="server" RenderMode="Inline" UpdateMode="Conditional">
  <ContentTemplate>

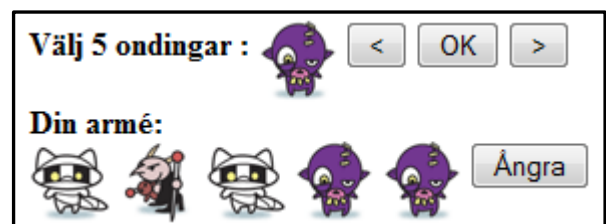
    <asp:Repeater runat="server" id="CharacterRepeater">
      <ItemTemplate>
        
      </ItemTemplate>
      <FooterTemplate>
        <asp:Button Text="Ångra" OnClick="Undo" runat="server"
          Visible="<##Characters.Count > 0 %>"/>
      </FooterTemplate>
    </asp:Repeater>

  </ContentTemplate>

  <Triggers>
    <asp:AsyncPostBackTrigger ControlID="Add" />
  </Triggers>

</asp:UpdatePanel>
</form>
```

ASP.NET



```

protected string[] Images = { "necrocat.gif", "voodoo.gif", "zombie.gif" };
protected List<string> Characters
{
    get { return (List<string>)ViewState["chars"]; }
    set { ViewState["chars"] = value; }
}
protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
        Characters = new List<string>();
}
protected void Nav(object sender, CommandEventArgs e)
{
    if (e.CommandName == "Prev")
        Viewer.ActiveViewIndex--;

    if (e.CommandName == "Next")
        Viewer.ActiveViewIndex++;
}
protected void Add(object sender, EventArgs e)
{
    if (Characters.Count < 5)
    {
        Characters.Add(Images[Viewer.ActiveViewIndex]);
        CharacterRepeater.DataSource = Characters;
        CharacterRepeater.DataBind();
    }
}
protected void Undo(object sender, EventArgs e)
{
    Characters.RemoveAt(Characters.Count - 1);
    CharacterRepeater.DataSource = Characters;
    CharacterRepeater.DataBind();
}

```

C#

ASP.NET AJAX från C#

Hittills har vi använt ASP.NET AJAX genom att deklarera allt i ASPX-filen utan att blanda in C# över huvudtaget. I många fall räcker detta, men det finns tillfällen då man exempelvis kan behöva uppdatera en updatePanel från programkoden beroende på något villkor, vilket man i C# gör med hjälp av metoden update() på panelen som skall uppdateras. För att en updatePanel skall kunna uppdatera sig från C# måste den också vara deklarerad med updateMode="Conditional", detta eftersom den annars ändå alltid kommer uppdatera sig.

Det kan finnas situationer då man behöver göra om en "vanlig"PostBack-kontroll till en asynkron dito i efterhand (även om det måste medges är ett ganska ovanligt fenomen). För att åstadkomma detta registreras kontrollen som en asynkronPostBack-kontroll i den scriptManager som hanterar JavaScript-delen på sidan.

Exemplet nedan har två stycken paneler som båda skriver ut aktuell tid när de uppdateras. Eftersom de har UpdateMode="Conditional" och saknar både Triggers och egna kontroller som postar tillbaka till sidan är det bara möjligt att uppdatera dem från programkoden.

Under panelerna finns två checkboxar som talar om vilken av de två som skall uppdateras. Knappen "Update" ligger inte i en updatePanel och blir av den anledningen inte heller registrerad som en kontroll som skall utföra en asynkron postback, detta görs istället i sidans Page_Load (detta är ett högst akademiskt exempel, det finns ingen teknisk anledning varför man skulle vilja registrera Button-kontrollen i C#-koden istället för på ASPX-sidan).

```
<form runat="server">
  <asp:ScriptManager runat="server" id="ScriptHandler"/>
  <asp:UpdatePanel ID="Timer1" runat="server" UpdateMode="Conditional">
    <ContentTemplate>Timer1: <%=DateTime.Now %></ContentTemplate>
  </asp:UpdatePanel>
  <asp:UpdatePanel ID="Timer2" runat="server" UpdateMode="Conditional">
    <ContentTemplate>Timer2: <%=DateTime.Now %></ContentTemplate>
  </asp:UpdatePanel>
  <asp:CheckBoxList ID="TimerSelector" runat="server" style="float:left"
    RepeatDirection="Horizontal">
    <asp:ListItem Value="Timer1">Timer 1</asp:ListItem>
    <asp:ListItem Value="Timer2">Timer 2</asp:ListItem>
  </asp:CheckBoxList>
  <asp:Button Text="Update" id="UpdateButton" runat="server" onClick="DoUpdate" />
</form>
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if ( ! IsPostBack)
        ScriptHandler.RegisterAsyncPostBackControl(UpdateButton);
}

protected void DoUpdate(object sender, EventArgs e)
{
    foreach (ListItem item in TimerSelector.Items)
    {
        if (item.Selected && item.Value == "Timer1")
            Timer1.Update();

        if (item.Selected && item.Value == "Timer2")
            Timer2.Update();
    }
}
```

C#

Timer1: 2009-09-26 21:27:11
Timer2: 2009-09-26 21:33:03

Timer 1 Timer 2

ASP.NET Uppgift 17-1

Den här relativt stora uppgiften går ut på att i ASP.NET skriva en enklare variant av kortspelet "Black Jack":

Spelet går ut på att komma så nära talet 21 som möjligt utan att bli "tjock" d.v.s. få över 21. Man måste dessutom ha bättre kort än dealern. Alla klädda kort är värda 10, Ess är 11 eller 1. Får spelaren 21 med två första kort kallas detta Black Jack.

Om både spelare och dealer har 17, 18 eller 19 så vinner dealern, har både spelare och dealer 20 eller 21 så blir det lika.

Spelaren kan alltid välja om han eller hon vill ha fler kort eller stanna. När spelarna valt klart drar dealern kort. Dealern måste stanna på 17 eller bättre och ta kort om det är under 17.

<http://sv.wikipedia.org/wiki/BlackJack>

Ovanstående regler skall användas, men spelet måste inte använda insatser eller någon form av poängräkning, huvudsaken är att det skall gå att spela ett parti mot datorn. Ej heller behöver de mer avancerade reglerna i Black Jack tillämpas.

Börja med att ladda ner <http://lix.hisvux.se/henrikvg/cs/cards.zip>

I filen cards.zip hittar du bilder på samtliga 52 kort och två C#-filer, Deck.cs och Card.cs. Om du inte känner för att skriva egen kod för att hantera kortleken kan du använda dessa två. Lägg till dem i ditt projekt och importera namnrymden kyAkademi.cs. Nästa sida visar hur klasserna i filerna fungerar och kan användas. Även om det nog inte behövs för att lösa uppgiften får du naturligtvis ändra hur mycket du vill i Card.cs och Deck.cs om du väljer att använda dem.

Uppgiften får lösas med alla till buds stående medel och se ut efter tycke och smak, så länge det framgår med tydlighet hur spelet fungerar, men naturligtvis skall spelet använda sig av en eller flera updatePanel (att göra så att spelet använder sig av ASP.NET AJAX är nog det absolut lättaste av hela uppgiften och kan med fördel implementeras i slutet)

Uppgiften fortsätter på nästa sida.

Card
+ Value: int + Color: Card.Colors

Deck
+ Size: int
+ Deck(): void + Reset(): void + TakeCard(): Card

<<enumeration>> Card.Colors
Hjärter Klöver Ruter Spader

En instans av klassen Deck simulerar en kortlek med blandade kort. Konstruktorn i Deck blandar korten automatiskt.

För att ta ett kort från toppen av kortleken används metoden TakeCard() som returnerar ett objekt av typen Card. Anropas TakeCard() trots att det inte finns några kort kvar i kortleken får man ett outOfCardsException. Egenskapen Size talar om hur många kort som finns kvar och metoden Reset() återför alla kort och blandar en ny lek.

Ett kort består av en färg (Card.Colors) och en valör (1-13), men tänk på att alla klädda kort i Black Jack skall räknas som 10 och att ess är 1 *eller* 11.

Nedan följer ett simpelt exempel som visar hur man skapar en ny kortlek och fyller en ListBox med alla 52 kort.

```
<form runat="server">
  <asp:ListBox runat="server" Rows="10" width="200" ID="AllCards">
  </asp:ListBox>
</form>
```

ASP.NET

```
using KyAkademien.Csharp;
protected void Page_Load(object sender, EventArgs e)
{
    Deck deck = new Deck();
    while (deck.Size > 0)
    {
        Card card = deck.TakeCard();
        AllCards.Items.Add(String.Format("{0:00} : color={1} ... value={2}",
                                         52 - deck.Size,
                                         card.Color,
                                         card.Value));
    }
}
```

C#

```
01 : color=Spader ... value=8
02 : color=Ruter ... value=1
03 : color=Hjärter ... value=1
04 : color=Klöver ... value=11
05 : color=Spader ... value=1
06 : color=Spader ... value=12
07 : color=Klöver ... value=3
08 : color=Klöver ... value=7
09 : color=Spader ... value=5
10 : color=Hjärter ... value=4
```

UpdateProgress

Kontrollen `updateProgress` används för att meddela användaren genom att exempelvis visa en text eller en animerad bild att en asynkron `PostBack` för tillfället är under bearbetning. Både kontrollen och konceptet bakom är relativt enkelt att förstå och använda; när en asynkron `PostBack` exekveras av användaren visar systemet (via JavaScript och CSS) en del av sidan som sedan göms undan igen när servern har svarat.

Innehållet i en `updateProgress` skrivs i en `<ProgressTemplate>` (se exemplet nedan) och kontroller där i är precis som i en `reapter`-kontroll inte tillgängliga när sidan kompileras. Vill man därför komma åt en kontroll i en `<ProgressTemplate>` måste man ge den ett ID och sedan använda sig av `FindControl`.

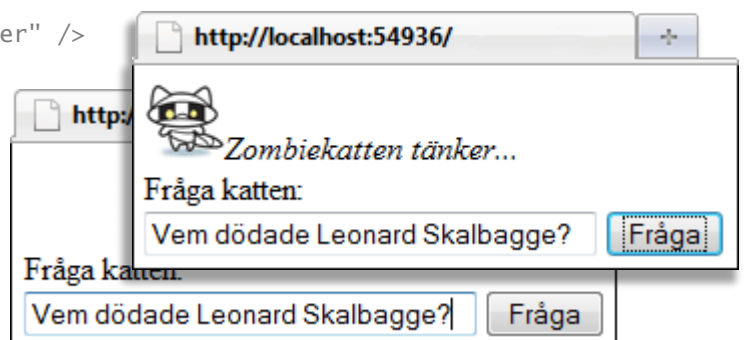
För att associera en `updateProgress` med en specifik `updatePanel` använder man sig av attributet `AssociatedUpdatePanelID`, som om det utlämnas innebär att `updateProgress` kommer gälla för *samtliga* paneler på sidan. Det går inte att ställa in så att en `updateProgress` utan `AssociatedUpdatePanelID` inte aktiveras av en `updatePanel`, vilket med andra ord innebär att en `updateProgress` utan någon specifik `updatePanel` associerad till sig alltid kommer aktiveras vid en asynkron `PostBack` - även för de paneler som har en annan "egen" `updateProgress`.

Standardinställningen är att en `UpdateProgress` aktiverar sig först en halv sekund efter att sidan har gjort en asynkron `PostBack`, alltså sker väldigt snabba asynkrona `PostBacks` utan att användaren får ett meddelande om att systemet håller på och uppdaterar sidan. För att förlänga eller förkorta den tiden använder sig `updateProgress` av attributet `DisplayAfter` som definieras i millisekunder (standardinställningen på en halv sekund skrivs alltså `DisplayAfter="500"`).

```
<form runat="server">
  <asp:ScriptManager runat="server" />
  <asp:UpdateProgress runat="server" DynamicLayout="false" DisplayAfter="0">
    <ProgressTemplate>
      
      <i>Zombiekatten tänker...</i>
    </ProgressTemplate>
  </asp:UpdateProgress>
  <asp:UpdatePanel runat="server">
    <ContentTemplate>
      Fråga katten:<br />
      <asp:TextBox runat="server" ID="Question" width="220"/>
      <asp:Button runat="server" Text="Fråga" OnClick="AnswerQuestion"/>
      <br />
      <asp:Label runat="server" ID="Answer" />
    </ContentTemplate>
  </asp:UpdatePanel>
</form>
```

ASP.NET

När användaren klickar på knappen sker en asynkron `postback` och innehållet i `<ProgressTemplate>` visas tills ett svar kommer tillbaka från servern. Attributet `DynamicLayout="false"` används för att sidan redan när den laddas in skall reservera plats för innehållet i `<ProgressTemplate>`.



Timer

En `Timer` används för att med jämna mellanrum göra en `PostBack` automatiskt, vanligen för att uppdatera en eller flera `updatePanel`. För utvecklaren kan `Timer` liknas vid en osynlig knapp som upprepade gånger klickar på sig själv. Precis som alla andra kontroller som gör en `PostBack` kan den registreras som en `Trigger` för en `updatePanel` och blir automatiskt en `Trigger` om den ligger i en panels `<ContentTemplate>`, såvida inte panelen har `childrenAsTriggers="False"`. Det är viktigt att notera att en `Timer` inte automatiskt gör en asynkron `PostBack` utan som alla andra `PostBack`-kontroller måste den först "registreras" som sådan.

`Timer` har bara två intressanta attribut; `onTick`, som utför en händelse precis som en `onClick` på en knapp varje gång den aktiveras, och `Interval` som talar om hur ofta kontrollen skall göra en `PostBack`. Värdet till `Interval` anges i millisekunder, alltså innebär exempelvis `Interval="2000"` att kontrollen skall posta tillbaka till sidan varannan sekund.

I Exemplet nedan uppdateras en `updatePanel` en gång i sekunden,

```
<form runat="server">
  <asp:ScriptManager runat="server"/>
  <asp:Timer runat="server" id="Ticker" OnTick="UpdateCountdown" Interval="1000" />
  <asp:UpdatePanel runat="server">
    <ContentTemplate>
      Du har <%= Seconds %> sekunder kvar innan Internet stängs av.
    </ContentTemplate>
    <Triggers>
      <asp:AsyncPostBackTrigger ControlID="Ticker" />
    </Triggers>
  </asp:UpdatePanel>
</form>
```

ASP.NET

```
DateTime Countdown
{
  get { return (DateTime)ViewState["countdown"]; }
  set { ViewState["countdown"] = value; }
}

protected int Seconds { get; set; }

protected void Page_Load(object sender, EventArgs e)
{
  if ( ! IsPostBack)
  {
    Seconds = 10;
    Countdown = DateTime.Now.AddSeconds(Seconds);
  }
}


protected void UpdateCountdown(object sender, EventArgs e)
{
  Seconds = Countdown.Subtract(DateTime.Now).Seconds;
  if (Seconds <= 0)
    Response.Redirect("about:blank");
}
```

C#

ASP.NET Uppgift 18-1

I den här uppgiften skall du skriva en databasdriven Webchat med ASP.NET AJAX. Uppgiften består av två stycken sidor, en login-sida och en sida för själva chatten. Lättast är nog att göra lite på båda innan du ger dig på att få dem att samspela.

För den här uppgiften behöver du två stycken databastabeller, en som håller reda på vilka som är på chatten (webchatUsers) och en som håller i själva konversationen (webChat),

WebChat			
	Column Name	Data Type	Allow Nulls
	ID	int	<input type="checkbox"/>
	InsertTime	datetime	<input type="checkbox"/>
	ChatText	varchar(250)	<input type="checkbox"/>
	Username	varchar(9)	<input type="checkbox"/>

WebChatUsers			
	Column Name	Data Type	Allow Nulls
	Username	varchar(9)	<input type="checkbox"/>
	LastSeen	datetime	<input type="checkbox"/>

ID-kolumnen skall vara *Identity* ("auto increment" på MySQLska) och kolumnerna med datetime som datatyp skall i *Default Value or Binding* ha värdet `getDate()`, vilket innebär att fältet automatiskt får aktuell tid och datum när en ny rad skapas.

Fältet `ChatText` innehåller en rad text som någon (`Username`) har sagt, såvida fältet `Username` inte är `null` - i så fall anses raden vara från "chatten" och används för att skriva ut vem som loggar in och loggar ut (se bilden på nästa sida).

Loginsidan

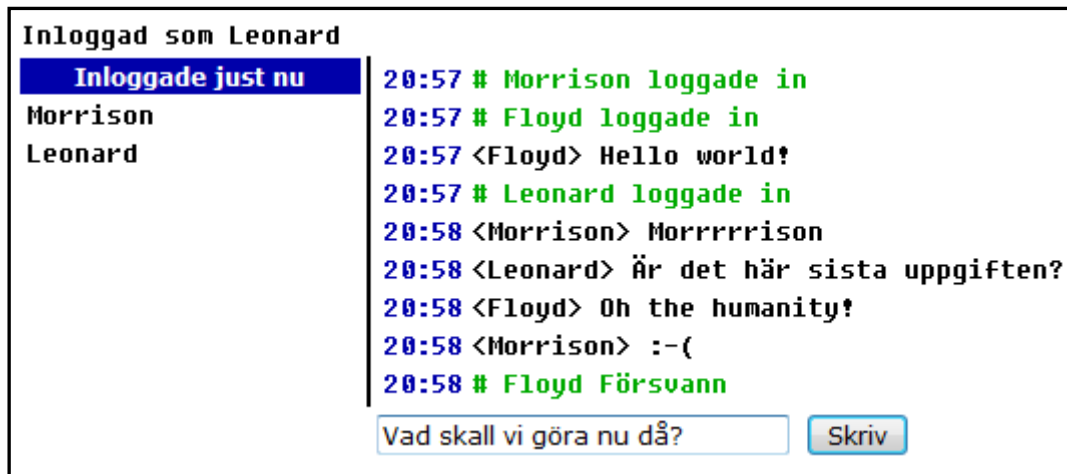
Utseendemässigt är loginsidan väldigt simpel, mycket mer än en textbox för användarnamn och en knapp med texten "Login" behövs inte. När användaren försöker logga in skall sidan kontrollera om användarnamnet finns i `webchatUsers` och om så är fallet anses det upptaget och användaren får ta och välja ett annat.

Finns namnet inte i `webchatUsers` skrivs det in och användaren förflyttas till chatten, men innan dess skall namnet skrivas till användarens session. Eftersom vi här inte går igenom ASP.NETs autentiseringsklasser måste vi själva hålla reda på vad användaren heter. Objektet `session` som heterar sessiondata fungerar på samma sätt som `viewState`, med skillnaden att innehåll i den överlever mellan olika sidor. Se slutet av kompendiet för ett exempel på hur `session` fungerar om det känns helt snurrigt idag.

Uppgiften fortsätter på nästa sida.

Chatsidan

Chatsidan är mer komplicerad och består av två stycken Repeater-kontroller, en för alla användare som är inloggade (vilket är ett aningen fånigt ordval i det här fallet) och en för själva konversationen. Båda dessa två skall ligga i var sin updatePanel då chatten skall fungera helt asynkront.



Bara ett exempel, din chat måste inte se såhär ful ut.

All text som skrivs skall sparas i databasen men bara de senaste 20 (eller ännu färre) raderna skall faktiskt visas,

```
SELECT TOP 20 * FROM WebChat ORDER BY ID DESC
```

SQL

Eller om du som i exemplet på bilden ovan istället vill ha det senast skrivna längst ner på sidan,

```
SELECT * FROM (SELECT TOP 20 * FROM WebChat ORDER BY ID DESC) AS wc ORDER BY ID ASC
```

SQL

När en person loggar in skall de inte mötas av en tom skärm utan istället direkt få de senaste 20 raderna presenterade. Förutom att chatten såklart uppdaterar sig om man skriver något i den skall den också uppdatera sig automatiskt var 10:e sekund med hjälp av en Trigger-kontroll, detta så att en person kan följa med i konversationerna utan att behöva skriva något själv.

För att formatera en DateTime-kolumn i en Repeater-kontroll så att den visar timmar och minuter som i exemplet ovan kan följande syntax användas:

```
<%# DataBinder.Eval(Container.DataItem, "InsertTime", "{0:HH}:{0:mm}") %>
```

ASP.NET

Uppgiften fortsätter på nästa sida.

Det absolut pilligaste i den här uppgiften är att hålla reda på vilka användare som är på chatten; en användare skall automatiskt loggas ut och användarnamnet tas bort från webChatUsers om han eller hon exempelvis stänger ner webbläsaren eller sin uppkoppling mot Internet. Samtidigt skall de som fortfarande är på chatten få ett meddelande om att personen inte längre finns kvar och listan med namn skall uppdateras.

För detta används kolumnen LastSeen i tabellen webChatUsers. Det fältet talar om när en användare senast uppdaterade sidan och skall uppdateras med aktuell tid varje gång sidans Trigger gör enPostBack.

När chatten uppdateras, antingen av en Trigger eller att det skrivs något, så skall de användare som inte uppdaterat sidan på 60 sekunder plockas bort,

```
SELECT Username FROM webChatUsers WHERE DATEDIFF(s, LastSeen, GetDate()) > 60
```

SQL - Plockar ut alla användare som inte uppdaterat sidan senaste minuten.

Använd en SqlDataReader och loopa igenom alla användare på chatten som skall tas bort. För varje användare som kommer försvinna skriver du en rad till tabellen webChat i stil med "Användare *användarnamnet* har lämnat" eller liknande innan du plockar bort personen från webChatUsers med en DELETE. Du måste öppna en ny SqlConnection som gör DELETE-kommandon, det är inte möjligt att återanvända en SqlConnection som är bunden till en öppen SqlDataReader.

Efter du har uppdaterat databaserna, alltså...

- 1 Uppdaterat användarens LastSeen
- 2 Plockat bort eventuella användare som inte längre är kvar på chatten
- 3 Lagt in eventuell ny text som användaren har skrivit till chatten

...så binder du sidans två Repeater-kontrollerna på nytt och uppdaterar panelerna.

APPENDIX

Request och Response

Request och Response är två objekt i ASP.NET som, precis som det låter, står för (klientens) förfrågan och (serverns) svar. Båda objekten skapas automatiskt när en sida laddas in och finns därför alltid tillgängliga. I Request-objektet hittar man saker som vilken sökväg som användes för att komma åt sidan, eventuella query-variabler, vilken webbläsare som användes och så vidare, medan Response talar om och hanterar hur servern svarar. Det lättaste sättet att se vad som finns i Request och Response är att sätta en breakpoint i koden och inspektera dem i exempelvis *Watch*-fönstret i editorn.

Nedanstående exempel använder Request-objektet för att skriva ut vilken IP och webbläsare som användaren har. I `Page_Load` kontrolleras det om query-variabeln "goto" är satt till "randomize", och i så fall slumpas användaren till en annan URL med hjälp av metoden `Response.Redirect`.

```
<form runat="server">
  Hej      <%=Request.UserHostAddress %>!<br />
  Du använder <%=Request.UserAgent %><br />
  <hr />
  Gå till... <a href="default.aspx?goto=randomize">någonstans</a>.
</form>
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request["goto"] == "randomize")
    {
        string url = "";

        switch ((new Random()).Next(0, 3))
        {
            case 0: url = "http://x25.se/"; break;
            case 1: url = "http://zombo.com/"; break;
            case 2: url = "http://icanhascheezburger.com/"; break;
        }

        Response.Redirect(url);
    }
}
```

C#

```
Hej 127.0.0.1!
Du använder Mozilla/5.0 (Windows; U; Windows NT 6.0; en-GB; rv:1.9.0.10) Gecko/2009042316 Firefox/3.0.
Gå till... någonstans.
```

Cookies

Cookies i ASP.NET är tämligen lätta att använda genom `Request.Cookies` och `Response.Cookies`,

```
<form runat="server">
  <asp:Label runat="server" ID="CookieTest" />
</form>
```

ASP.NET

```
protected void Page_Load(object sender, EventArgs e)
{
    if (Request.Cookies["LastVisit"] == null)
    {
        CookieTest.Text = "Du har inte varit här tidigare.";
    }
    else
    {
        CookieTest.Text = "Välkommen tillbaka, du var här senast: "
            + Request.Cookies["LastVisit"].Value;
    }

    // Skapar en ny cookie, lagrar aktuell tid i 100 dagar
    HttpCookie cookie = new HttpCookie("LastVisit");
    cookie.Value = DateTime.Now.ToString();
    cookie.Expires = DateTime.Now.AddDays(100);

    // ...och lägger till eller uppdaterar den i Response.Cookies
    Response.Cookies.Add(cookie);
}
```

C#

Sessioner

Sessionshantering sköts till stort av ASP.NET automatiskt och styrs genom konfigurationsfilen `web.config`. Det som brukar ställa till störst huvudbry är hur sessionerna är lagrade, för som standard lagras sessionerna *InProc*, eller med lättare termer, i ASP.NET-processens minne. Det innebär att varje gång ASP.NET startar om sin process, vilket vanligen händer flera gånger om dagen automatiskt, så försvinner samtliga pågående sessioner. Detta är naturligtvis oacceptabelt för exempelvis en webbshop, då alla för tillfället inloggade kunder loggas ut och får sina varukorgar tömda. Lösningen brukar vara att ändra *SessionState* i `web.config` till att spara ner sessionsdatan i SQL Server istället, som gör att de då överlever även om så webbservern startas om. Detta kräver att man installerar ASP.NETs SQL-tabeller för sessionshantering, vilket vi inte kommer ta upp här.

`Session`-objektet ligger varken i `Response` eller `Request` utan direkt på `Page`-klassen, vilket gör att man kommer åt det från sidorna genom att bara skriva `Session`. Användandet av `Session` fungerar på det hela stora exakt som vi är vana vid att använda `ViewState`-objektet, den stora skillnaden är naturligtvis att `Session` "överlever" mellan olika sidor vilket ett objekt sparad i `ViewState` inte gör.

Exemplet på nästa sida använder `Session`-objektet för att spara ner ett objekt av typen varukorg, en väldigt simpel klass där man kan stoppa i och plocka ut strängar*. Precis som med `ViewState` måste vi typecasta tillbaka vårt objekt till rätt datatyp varje gång vi skall använda det.

* På grund av platsbrist är klassen för varukorgen väldigt ofullständig.

```

<form runat="server">
  <asp:Button Text="Skapa ny varukorg" runat="server" OnClick="CreateNewVarukorg"/>
  <hr />
  Skriv något att lägga till i korgen:
  <asp:textbox ID="StuffToAdd" runat="server" />
  <asp:Button runat="server" Text="Lägg till" OnClick="AddItem"/>
  <br />
  <asp:Repeater runat="server" ID="ViewContent">
    <ItemTemplate>
      <asp:Button Text="Ta bort" runat="server" OnCommand="RemoveItem"
        CommandArgument="<## Container.DataItem %>" />
      <## Container.DataItem %><br />
    </ItemTemplate>
  </asp:Repeater>
</form>

```

ASP.NET

```

public partial class SessionTest : System.Web.UI.Page
{
  class Varukorg
  {
    public List<string> Stuff = new List<string>();
  }

  public void Page_Load(object sender, EventArgs e)
  {
    // Skapar en ny varukorg så fort användaren surfar in på sidan
    // om han eller hon inte har en skapad sedan tidigare.
    if ( ! IsPostBack && Session["Varukorg"] == null)
      Session["Varukorg"] = new Varukorg();
  }

  protected void RemoveItem(object sender, CommandEventArgs e)
  {
    Varukorg korg = Session["Varukorg"] as Varukorg;
    korg.Stuff.Remove((string)e.CommandArgument);
    UpdateRepeater();
  }

  protected void AddItem(object sender, EventArgs e)
  {
    Varukorg korg = Session["Varukorg"] as Varukorg;
    korg.Stuff.Add(StuffToAdd.Text);
    UpdateRepeater();
  }

  protected void CreateNewVarukorg(object sender, EventArgs e)
  {
    Session["Varukorg"] = new Varukorg();
    UpdateRepeater();
  }

  private void UpdateRepeater()
  {
    Varukorg korg = Session["Varukorg"] as Varukorg;
    ViewContent.DataSource = korg.Stuff;
    ViewContent.DataBind();
  }
}

```

C#