# Esame Microsoft 70-256

Riccardo Pietrucci

24 gennaio 2008

## 1 Fondamentali

### 1.1 I ruoli

La comunicazione dal browser al Web Server coincide con l'oggetto *Request*. Viceversa, la comunicazione dal Web Server al browser coincide con l'oggetto *Response*.

Una *request* è un comando simile al seguente[1]:

```
GET /default.aspx HTTP/1.1
Host: www.northwindtraders.com
```

Dove la prima parte è il metodo (anche detto verb o command) seguito da un URL e dalla versione di HTTP da utilizzare, mentre la seconda parte è il nome dell'host, utile se il Web Server mantiene più siti web. I metodi più comuni sono GET, HEAD, POST, CONNECT, PUT/DELETE, OPTIONS, LOCK/UNLOCK. Una *response* è espressa nel seguente formato:

```
HTTP/1.1 200 OK
Server: Microsoft-IIS/6.0
Content-Type: text/html
Content-Length: 38
<html><body>Hello, world.</body><html>
```

Il reinvio di una pagina a sé stessa[2] è un *PostBack* verso il server che accetta sia metodi POST che GET, in cui le pagine ASP .NET possono verificare la proprietà IsPostBack. Per verificare errori è possibile utilizzare uno sniffer di rete, come *Microsoft Network Monitor* o un tool come *Telnet*[3].

### 1.2 Creare un sito web

Una pagina web ASP .NET (*Web Form*) è costituita da direttive, script e layout. Lo script dovrebbe solo gestire codice lato client, ma nel caso in cui si voglia gestire anche codice server si usa la forma:

```
<script runas="server">
private void SayHi(object sender, EventArgs args) {
    Response.Write("Hello " + txtName.Value);
}
</script>
```

Questa forma viene chiamata programmazione *in-line*. In contrapposizione, la programmazione *code-behind* consente di applicare il pattern MVC, suddividendo visualizzazione (`*.aspx`) dalla logica (`*.aspx.cs`). Il rapporto tra i due stili è il seguente:

> dati due insiemi $A$ e $B$, dove $A$ è il team che si occupa della grafica e $B$ è il team che si occupa della programmazione, $A$ è il grafico HTML, $B$ è il programmatore code-behind, $A \cap B$ è il programmatore in-line

Ma il programmatore in-line non è un esperto in entrambi i settori: è semplicemente una figura con le conoscenze necessarie a scrivere quel codice utile a rappresentazioni grafiche programmatiche (ad esempio un tree grafico). Non è quindi una figura che si interfaccia ad un database o esegue controlli e/o operazioni complesse.

In entrambi gli stili *la compilazione è dinamica*: viene compilata la pagina al momento della sua prima richiesta. In caso di modifiche verrà ricompilata quando richiesto. Se presenti errori non verrà eseguita esclusivamente la pagina causa di errori.

Si possono utilizzare *cartelle speciali* non-browsable, tra cui `App_Data` che contiene informazioni sul database, `App_Code` che contiene il codice sorgente delle classi e degli oggetti business che si vuole compilare nel progetto, `Bin` che contiene assembly compilati (file dll) che il progetto referenzia in modo automatico.

### 1.3 La configurazione

I siti web sono configurabili tramite una gerarchia di file di configurazione XML:

**machine.config globale** configurazione radice per ogni tipo di applicazione, collocata in:

```
%WINDIR%\Microsoft.NET\Framework\%VERSION%\Config
```

**web.config globale** configurazione per ogni sito web, collocata anch'essa in:

```
%WINDIR%\Microsoft.NET\Framework\%VERSION%\Config
```

**web.config locale in /** configurazione per uno specifico sito web, collocata

**web.config locale in /application** configurazione per una specifica applicazione web

**web.config locale in /application/subsite** configurazione per una sottodirectory di una specifica applicazione web

Per cancellare le proprietà ereditate da un file di configurazione sopra la gerarchia si utilizza `<clear/>`. Il runtime costruisce una cache delle impostazioni effettive di un sito web. Per modificare la configurazione si possono utilizzare tecniche classiche di modifica xml (a mano, in modo programmatico, ...) oppure utilizzare MMC o il Web Site Administration Tool, accessibile tramite VisualStudio nel menu Web Site | ASP.NET Configuration.

Tra le configurazioni disponibili vi è il *trace*, che consente di rilevare e diagnosticare problemi e di esplorare l'utilizzo delle risorse di un sito web. La visualizzazione delle informazioni può essere accodata a ogni pagina web, o archiviata in una pagina virtuale appositamente creata, localizzata in:

```
http://server/application/trace.axd
```

[1] i comandi sono analizzabili tramite telnet impostando le opzioni (`set`) `localecho` e `crlf`

[2] ad esempio come risposta all'invio di una form per avvisare: "username già esistente"

[3] una volta avviato impostare *set localecho* e *set crif*

Tra i caratteri speciali, la *tilde* indica la cartella dell'applicazione web corrente. Per tradurla nel suo path assoluto si utilizza il metodo *HttpServerUtility.MapPath*. L'istanza HttpServerUtility è ottenibile tramite *Response.Server* o *HttpContext.Current.Server*.

# 2 Aggiungere e configurare controlli server

La sfida per uno sviluppatore nel creare un sito web consiste nell'offrire la sensazione all'utente che il browser e il server siano una cosa unica attraverso una comunicazione bidirezionale, difficile da ottenere. ASP.NET aiuta lo sviluppatore tramite i server control, che forniscono automaticamente la comunicazione con il web server a seguito di un evento scatenato. Sono facilmente identificabili tramite l'attributo *runat="server"* e mantengono automaticamente lo stato tra le chiamate verso il server tramite *ViewState*. ViewState è implementato utilizzando nella form in cui avviene lo scambio di dati un campo nascosto "__VIEWSTATE" ed inviando i dati da/verso il client:

```
<input type="hidden" name="__VIEWSTATE"
       value="dDw4OTE1NzQzMjt0PDtsPGk8M..." />
```

è facile pensare che la stringa sia crittografata, ma si tratta di una semplice codifica in Base64. In pagine con parecchi web controls, la value inizierà ad avere una certa lunghezza, andando così ad influire pesantemente nelle prestazioni della pagina (server round-trip). Bisogna quindi cercare di utilizzare il ViewState solamente nella pagine web in cui ne abbiamo effettivamente bisogno impostando la proprietà *EnableViewState* a false alla direttiva @Page o ai controlli. Il value di un ViewState è dato quindi dall'unione degli stati di molteplici controlli e per verificarne il singolo contributo si può utilizzare il tracing nella sezione "Control Tree".

Tutti i server control e le web page derivano dalla classe base *Control*. Un Control ha il seguente ciclo di vita[4]:

***PreInit***

***Init*** inizializza ogni controllo figlio (ad esempio un DataGrid inizializza tutte le celle)

***Load*** azioni in comune a ogni richiesta (anche a seguito di un postback[5]). Viene eseguita prima dell'esecuzione degli eventi, come la pressione di un pulsante (specialmente vero in AJAX)

***PreRender*** cambiamenti all'ultimo minuto prima di salvare il ViewState a seguito di un evento PostBack. Viene eseguito dopo l'esecuzione degli eventi (come in AJAX)

**Render** genera l'HTML, DHTML e script necessari per mostrare il controllo al browser

***Unload*** rilascia ogni risorsa managed

**Dispose** rilascia ogni risorsa unmanaged

Gli eventi in corsivo possono essere catturati senza sottoscrizione se si imposta a true la proprietà AutoEventWireup nella direttiva @Page e si implementa un metodo con nome *Page_Event*:

```
private void Page_Init(object sender,
              EventArgs e) { }
```

[4]e quindi lo condividono anche i server control e le web page

[5]a meno che non sia stata gestita la condizione *Page.IsPostBack*

Un controllo viene normalmente incluso in modo "statico" nell'aspx, ma può anche essere aggiunto dinamicamente nel codice tramite *TemplateControl.LoadControl*. Una Page deriva da TemplateControl e supporta quindi tale metodo.

Esistono due tipi diversi di server control, che derivano dalla classe Control:

**HtmlControl** sono come i normali tag HTML ma contengono inoltre l'attributo *runat="server"* che va sostituito al posto del tag "action". Normalmente il rendering coincide con un singolo tag html

**WebControl** rappresentano elementi complessi il cui rendering coincide in genere con vari tag HTML ed eventuale codice JavaScript browser-indipendent. Espongono un modello consistente con le convenzioni WinForm. Nella pagina web il tag è espresso con:

```
<asp:textbox attributes runat="server" />
```

Lo scopo principale di un HTMLControl è di consentire una semplice conversione di vecchi siti web in ASP.NET o di abilitare JavaScript custom lato client.

Esistono diverse proprietà impostabili, tra cui Style, Visible[6]. Sono impostabili direttamente nella pagina HTML, per via programmatica o attraverso il Design view di Visual Studio.

Alcuni server control provocano un PostBack automatico quando un evento si verifica (Click in un button) e gli eventi sono chiamati *eventi immediati*. Altri server control (TextBox) non prevedono PostBack automatico e gli eventi sono chiamati *eventi posticipati*. Gli eventi posticipati vengono attivati da un PostBack ma sono eseguiti prima degli eventi che hanno causato il PostBack. Per cambiare un evento posticipato a un evento immediato è necessario impostare nel control la proprietà *AutoPostBack="true"*.

Alcuni server control (come WebPage) sono *INamingContainer*, cioè sono contenitori per altri server control e dichiarano quindi un namespace univoco per impedire ambiguità nei nomi. Questo significa che possiedono una collezione *Controls*, ottenibili dinamicamente tramite il metodo *FindControl*. Questo può sembrare poco utile, poiché è possibile utilizzare l'elemento tramite il suo ID, ma risulta comunque necessario in situazioni dinamiche, come nel caso di GridView che crea dinamicamente i controlli figlio necessari.

Tra i server control più utilizzati troviamo i controlli *Label* per testo dinamico, *Literal* per testo statico, *TextBox* per testo immesso dagli utenti, *CheckBox* per selezioni true o false, *CheckBoxList* per gruppi di CheckBox, *RadioButton* per scelte booleane esclusive, *RadioButtonList* per gruppi di RadioButton, *Button* per un pulsante push. Il controllo Button può essere un pulsante Submit o un pulsante Command nel caso si specifichi la proprietà *CommandName* e può prevedere o meno validazione a seconda del valore di *CausesValidation*, utile con pulsanti Reset ed Help.

Per visualizzare stringhe di testo esattamente come sono scritte senza incorrere nel pericolo che non vengano formattate e presentate correttamente all'utente finale si utilizza *Server.HtmlEncode* e *Server.HtmlDecode*: ad esempio, data la stringa:

```
°°° Ciao °°°
```

questa verrà tradotta in:

```
&deg;&deg;&deg; Ciao &deg;&deg;&deg;
```

[6]se il valore è false l'elemento non compare affatto nel rendering

# 3 Controlli Server Specializzati

## 3.1 Controlli Web Server Specializzati

Il controllo *Literal* è simile al Label e viene reso come un tag `<span>`. Attraverso la proprietà *Mode* si può impostare la gestione del contenuto:

**PassThrough** nessuna modifica

**Encode** i tag non hanno effetto

**Transform** convertito secondo il linguaggio di markup corrispondenti al browser (HTML, XHTML, WML, ...)

Il controllo *Table* consente di aggiungere dinamicamente *TableRow*, composte da *TableCell*. E' utile solo se facenti parte di controlli custom, altrimenti non è mai consigliabile né con tabelle statiche (si dovrebbe utilizzare il semplice tag HTML `<table>`) né con tabelle dinamiche (i dati vengono ricreati ad ogni PostBack). Con dati dinamici conviene infatti utilizzare altri controlli come *Repeater*, *DataList* o *GridView*.

Il controllo *Image* consente di mostrare immagini e vengono trattate come richieste separate da parte del browser. La proprietà *GenerateEmptyAlternateText* è utile per motivi di accessibilità: se `true` le immagini puramente decorative vengono ignorate dai browser text-based. Per cliccare su un'immagine si può utilizzare la sottoclasse di Image *ImageButton* che consente inoltre di ottenere le coordinate xy e corrisponde al tag `<input type="image">`. Per definire delle regioni (classe *HotSpot*) nell'immagine si utilizza la sottoclasse di Image *ImageMap*, che viene resa come `<img usemap="#myMap">`. Possono esistere diversi hotspot che si sovrappongono: in questo caso prende precedenza l'hotspot definito per primo (pupilla, orbita, occhio, viso, ...). La classe HotSpot prevede diverse implementazioni: *CircleHotSpot*, *RectangleHotSpot*, *PolygonHotSpot*. La proprietà *HotSpotMode* consente di specificarne il comportamento e può essere definito un comportamento predefinito nell'oggetto ImageMap e uno specializzato nell'oggetto HotSpot e consente di definire le seguenti modalità:

**NotSet** non è impostato nulla. In HotSpot significa che eredita le impostazioni specificate in ImageMap. Se anche ImageMap imposta NotSet allora il comportamento di default è Navigate

***Navigate*** (default) naviga verso *NavigateURL* quando cliccato

**Postback** provoca un postback al server restituendo una proprietà *PostBackValue* al web server

**Inactive** nessun comportamento. Utile per creare una regione hotspot inattiva all'interno di una attiva. In questo caso la regione inattiva deve essere specificata *prima* di realizzare quella attiva

Il controllo *Calendar* consente di mostrare e gestire un calendario tramite i due eventi *SelectionChanged* (selezione di una data, ottenibile con la proprietà *SelectedDates*) e *VisibleMonthChanged* (selezione di un differente mese da visualizzare). Il rendering HTML coincide con il tag `<table>`. Inoltre, l'evento *DayRender* consente di aggiungere testo o controlli al giorno del mese in corso di rendering. La proprietà *SelectionMode* consente di determinare se consentire selezioni sia nulle (*None*) che di date (*Day*) che di settimane (*DayWeek*) che di mesi (*DayWeekMonth*). Calendar non è serve solo a selezionare delle date, ma anche a costruire uno schedule per mostrare appuntamenti o ricorrenze all'interno di un calendario. Per realizzare questo comportamento è sufficiente rendere il controllo abbastanza ampio (in `Page_Load`) da contenere un controllo Label da aggiungere alla collezione *Cell.Controls* nel gestore dell'evento DayRender.

Il controllo *FileUpload* genera il tag `<input type="file">`. Non provoca un PostBack ma è necessario che venga invece provocato da un controllo differente come Button. Il postback comporta l'upload del file sul server tramite invio POST e il page code di risposta non viene eseguito nel server finquando il file non è stato trasferito nel server. Per accedere al file inviato si possono utilizzare le seguenti proprietà:

**FileBytes** come array di byte

**FileContent** come flusso

**PostedFile** come *HttpPostedFile*, che contiene le proprietà *ContentType* e *ContentLength* utili per verificare il file

Dopo la verifica (da effettuare lato server), che può comprendere anche un controllo di eventuali virus, si procede con il metodo FileUpload.SaveAs. Di default in `web.config` l'elemento: `<httpRuntime> <requireRootedSaveAsPath>` value true, il che significa che è necessario fornire un path assoluto al metodo SaveAs tramite l'utilizzo di MapPath e della tilde. La massima dimensione è specificata nell'elemento `<httpRuntime> <MaxRequestLength>`.

Il controllo *Panel* consente di contenere vari controlli per mostrarli o nasconderli come gruppi, generando un corrispondente elemento HTML `<div>`. Consente inoltre di impostare uno sfondo, l'allineamento o il pulsante da cliccare nel caso di pressione del tasto Enter (DefaultButton).

Il controllo *MultiView* consente di interscambiare le varie *View* che contiene[7]. Non trova corrispondenza in tag html, ma sono utili solo per un comportamento lato server. Risulta utile per creare Wizard o per applicazioni che prevedono diverse schermate per dispositivi mobili (in questo caso ha la stessa funzionalità del controllo mobile *Form*).

Il controllo *Wizard* utilizza le MultiView per personalizzare in varie parti il processo di wizard. Consente di mostrare una serie di controlli *WizardStep* che rappresentano raggruppamenti logici o passi e permette di definire la logica di navigazione dei passi. Ogni WizardStep consente di definire la proprietà *StepType*:

**Auto** visualizzazione basaa sulla sua posizione all'interno dei vari step

**Complete** l'ultimo step: non viene mostrato alcun pulsante di navigazione

**Finish** l'ultimo step che prevede raccolta di dati

**Start** il primo pulsante: disabilita il pulsante Previous, che non ha senso in questo punto

**Step** uno step tra Start e Finish, in cui sia Previous che Next devono comparire

[7] comportamento simile a delle Swing di Java

La proprietà Wizard.*ActiveStepIndex* consente di specificare lo step attivo nel wizard. E' necessario specificare lo step iniziale nel Load della pagina.

Il controllo *Xml* consente di:

- visualizzare un documento XML. Si può specificare il percorso del file (DocumentSource) o il contenuto XML (DocumentContent)
- eseguire una trasformazione XSL. Si può specificare il percorso del file (TransformSource) o il contenuto XML (TransformContent). Si possono passare argomenti alla trasformazione tramite *TransformArgumentList*.

Normalmente viene specificato un documento XML da visualizzare che viene sottoposto a una trasformazione XSL per adattare la visualizzazione a quanto desiderato (si impostano quindi solitamente le proprietà DocumentSource e TransformSource).

# 4 Controlli Web Server Data-Bound

## 4.1 Concetti introduttivi

I controlli Data-bound (*BaseDataBoundControl*) sono controlli che prevedono di associarsi o connettersi a dati. Sono classificati come:

**semplici** controlli che ereditano da *ListControl* e da *AdRotator*

**compositi** controlli che ereditano da *CompositeDataBoundControl* come GridView, DetailsView e FormsView

**gerarchici** controlli che ereditano da *HierarchicalDataBoundControl*

Ogni controllo data-bound[8] prevede il metodo *DataBind* per trasferire i dati specificati nel controllo. Nel caso di controlli compositi il metodo discende ricorsivamente nei figli richiedendo il corrispondente DataBind.

Ogni controllo data-bound consente di specificare la *sorgente* di dati tramite *DataSource* (nel code-behind) o tramite *DataSourceID* (nel markup). L'oggetto specificato nella proprietà DataSource è tipicamente un'istanza di una classe che implementa IEnumerable, IListSource, IDataSource o IHierarchicalDatasource. Tipicamente le sorgenti di dati disponibili ereditano da DataSourceControl o HierarchicalDataSourceControl e implementano internamente sia IDataSource che IListSource:

**AccessDataSource** binding con Microsoft Access

**SqlDataSource** binding tramite ODBC, OLEDB, SQL Server, Oracle

**XmlDataSource** binding a un file XML. Se il nodo radice dell'xml non è desiderato si utilizza l'espressione *XPath* per specificare da quale punto iniziare a ottenere l'xml. Ad esempio:

```
<asp:XmlDataSource
        ID="XmlDataSource" runat="server"
        DataFile="~/App_Data/MenuItems.xml"
        XPath="/MenuItems/*"/>
```

[8] si potrebbe leggere come: la classe BaseDataBoundControl...

**ObjectDataSource** binding a un oggetto. Utile per connettere oggetti business middle-tier o oggetti DataSet nella directory `Bin` o `App_Code`. Oltre a selezionare la classe è necessario specificare i metodi da utilizzare per l'esecuzione di operazioni select, insert, update e delete. L'operazione select deve restituire un singolo oggetto che implementa IEnumerable, IListSource, IDataSource o IHierarchicalDatasource

**SitemapDataSource** binding al site navigation tree

I *template control* sono controlli che di default non hanno alcuna UI ma forniscono semplicemente un meccanismo per eseguire il binding: è lo sviluppatore che fornisce la UI tramite template inline dotati di elementi come HTML o DHTML. Esempi di tali controlli sono GridView, DetailsView e FormView e possono essere programmati i seguenti template:

**HeaderTemplate** intestazione (opzionale)

**FooterTemplate** piè di pagina (opzionale)

**ItemTemplate** per ogni riga

**AlternatingItemTemplate** per ogni riga pari (opzionale). Se specificato viene usato ItemTemplate solo per le righe dispari

**SelectedItemTemplate** la riga che è stata selezionata

**SeparatorTemplate** il separatore tra una riga e l'altra

**EditItemTemplate** riga che è nella modalità edit

Per inserire i dati nei template si usa il metodo *DataBinder.Eval* che tramite Reflection esegue una ricerca delle proprietà del tipo specificato nella proprietà *DataItem*. Ad esempio per inserire il binding alla proprietà Car.Vin:

```
<%# Eval("Vin") %>
```

Il metodo Eval consente inoltre di inserire un secondo parametro per specificare il formato della stringa. Il metodo Eval è a una via ed è di sola lettura. E' utile per template come ItemTemplate ma non in EditItemTemplate: in questo caso i campi non sarebbero modificabili. E' possibile utilizzare in tal caso il metodo *DataBinder.Bind* a due vie, modificabile.

## 4.2 Controlli semplici

La classe *ListControl* contiene le funzionalità comuni per le sue sottoclassi:

**DropDownList** lista a tendina con selezione singola

**ListBox** lista estesa con selezione singola o multipla, a seconda del valore di *SelectionMode*

**RadioButtonList** lista di elementi a selezione singola

**CheckBoxList** lista di elementi a selezione multipla

**BulletedList** lista numerata o meno, coincidente con `<ol>` e `<ul>`, a seconda del valore di *BulletStyle*

In particolare tutti i ListControl contengono una collezione di oggetti *ListItem*, ognuno avente un *Text* visualizzato e un *Value* utilizzato internamente al sistema, inviato al momento del postback. Gli oggetti ListItem possono essere aggiunti o in modo programmatico o specificando le proprietà *DataSource*, *DataMember*, *DataTextField*, *DataValueField* e (opzionalmente) *DataTextFormatString*. Al momento della selezione di un elemento viene attivato l'evento SelectedIndexChanged e si può esaminare l'elemento selezionato tramite le proprietà SelectedIndex, SelectedItem o SelectedValue. Infine, la proprietà *AppendDataBoundItems*, se vera, consente di concatenare gli elementi derivanti dal binding con quelli già presenti in memoria. Nel caso di selezione singola è sufficiente analizzare *SelectedValue* (o simile), mentre nel caso di selezione multipla bisogna iterare in tutti gli elementi e verificare il valore di *Selected*.

La classe *AdRotator* consente di visualizzare annunci in modo casuale (tipicamente banner o tips) e di controllarne varie impostazioni tramite un file xml, tra cui la frequenza di visualizzazione (*Impressions*). Poiché i banner contengono script è necessario seguire le seguenti linee di sicurezza:

- validare gli script che caratterizzano gli annunci
- collocare gli annunci nella cartella `App_Data`, non esplorabile dai browser
- per la configurazione degli annunci su script XML, utilizzare l'estensione `.config`, non accessibile dai browser
- impostare i permessi degli annunci nell'account ASP.NET come read-only

### 4.3 Controlli compositi

I controlli *CompositeDataBoundControl* implementano INamingContainer[9] e consentono quindi di comporre vari controlli. Le classi che ereditano tale classe base sono GridView, DetailsView e FormsView.

**GridView** mostra dati in forma tabulare e ne consente ordinamento, paginazione[10] e modifiche. Contiene varie righe *GridViewRow*, colonne *DataControlField* e celle *DataControlFieldCell*. Anche se le celle *sono contenute* nelle righe, la loro *inizializzazione* avviene tramite le colonne, nel metodo DataControlField.*InitializeCell*. Per la formattazione si utilizzano gli stili, che obbediscono a regole gerarchiche[11]. Si può assegnare uno stile utilizzando l'evento *RowCreated*. Se invece è necessario assegnare uno stile basandosi sul valore assegnato alle celle, si utilizza l'evento *RowDataBound*.

**DetailsView** mostra i valori di una singola tupla e consente di modificare, *cancellare e aggiungere* righe (queste ultime non permesse in GridView). Se la proprietà *AllowPaging* è vera allora può essere utilizzata anche per navigare nell'intera sorgente di dati, e non nella singola tupla: questo consente di evitare l'utilizzo combinato dei controlli GridView, ListBox o DropDownList. A differenza di GridView, non supporta l'ordinamento. Anche in questo caso sono supportati gli stili, allo stesso modo di GridView.

[9] come già visto nel par. 2

[10] la paginazione divide una tabella in più pagine web per facilitarne la fruibilità

[11] Al primo livello: *GridView*. Al secondo livello: *Header*, *Footer*, *Row*. Man mano sempre più annidati in Row troviamo *AlternatingRow*, *SelectedRow*, *EditRow*, *Column*

**FormsView** come in DetailsView consente di visualizzare una singola tupla, ma consente inoltre di impostare template personalizzati. Abilita una grande flessibilità al costo di un maggiore sforzo per applicare alcuni stili impostabili in modo molto semplice in GridView e DetailsView.

### 4.4 Controlli gerarchici

I controlli HierarchicalDataBoundControl consentono di formattare dati in modo gerarchico. Le classi che ereditano tale classe base sono TreeView e Menu.

**TreeView** mostra dati sottoforma di albero gerarchico, utile per filesystem, indici. Se viene assegnato come sorgente di dati un SiteMapDataSource allora si può realizzare un site navigation. Un nodo *TreeNode* può essere *parent* o *child*. Un nodo child particolare è il nodo *leaf* e un nodo parent particolare è il nodo *root*. Un tipico TreeView possiede un solo nodo root, ma è possibile aggiungere più nodi root (come dire: ho un nodo root invisibile). Ogni TreeNode possiede un *Text* visualizzato e un *Value* effettivamente utilizzato dal sistema. Se un TreeNode possiede la proprietà *NavigateUrl* allora il nodo è di tipo navigazione, altrimenti è di tipo selezione. Per passare la sorgente di dati è necessario che il TreeView la "capisca" (chi è Text? chi è Value?...): TreeView contiene oggetti *TreeNodeBinding* che definiscono il collegamento (via Reflection) tra il data source specificato e i TreeNode.

**Menu** mostra dati sottoforma di un menu di sistema. Spesso utilizzato con data source SiteMapDataSource per consentire la navigazione del sito. Analogamente a TreeView contiene contiene oggetti *MenuItemBinding* che definiscono il collegamento tra il data source specificato e i *MenuItem*

**SiteMapPath** mostra una struttura gerarchica nella forma:

```
Home -> Visual Studio -> Support
```

utile da applicare nell'intestazione della pagina.

## 5 Persistenza dei dati

La persistenza viene gestita tramite le classi di ADO.NET, divise in due categorie: *connesse* e *disconnesse*

### 5.1 Classi ADO.NET disconnesse

Le classi disconnesse consentono di assegnare valori ad un'applicazione web senza essere necessariamente connessi. La classe fondamentale è la *DataTable*, composta da *DataColumn* (metadati o schema) e *DataRow* (dati) e *PrimaryKey* (riferimento a $n$ DataColumn). I DataColumn consentono di impostare varie proprietà come[12]: *DataType* (String), *MaxLength* (-1), *Unique* (false), *AllowDBNull* (true), *Caption* (ColumnName).

Ogni DataRow contiene tre *DataRowVersion*: *Original*, *Current* e *Proposed*. Durante la modifica si opera con la Proposed e al termine di tale modifica (metodo *AcceptChanges*) Proposed passa a Current, e Current passa a Original. In questo modo è possibile ottenere la copia precedente di tale riga per supportare un semplice undo, limitato però a un singolo elemento di cronologia. Con l'undo (metodo *RejectChanges*) il rollback ripristina Original a Current, ma viene completamente perso il precedente Original (non è possibile eseguire un redo). Riassumendo, il RejectChanges consente di ripristinare lo stato

[12] tra parentesi il suo default

assunto da AcceptChanges a un solo livello e non è previsto un redo. Inoltre, la versione *Default* restituisce Proposed se esiste, altrimenti Current. Nel caso in cui venga richiesto una versione inesistente viene lanciata un'eccezione (meglio verificare *HasVersion*).

Le DataRow possono avere uno dei seguenti *RowState*:

**Detached** riga creata ma non aggiunta

**Added** riga aggiunta

**Unchanged** riga senza alcun cambiamento dall'ultima esecuzione di AcceptChanges

**Modified** riga modificata dall'ultima esecuzione di AcceptChanges

è possibile impostare esplicitamente il RowState tramite i metodi di DataRow *SetAdded* e *SetModified*, utile per memorizzare i dati in un supporto differente da quello di default.

Per aggiungere righe si usa in modo appropriato[13] DataTable.*Add*, per importarle da un'altra DataTable si usa *ImportRow* e per aggiornarle si usa DataTable.*Load*, in cui si può specificare una delle seguenti *LoadOption*:

**OverwriteRow** sovrascrive le DataRowVersion *Original* e *Current*. Cambia sempre il RowState a *Unchanged*

**PreserveCurrentValues** (default) sovrascrive le DataRowVersion *Original*, ma lascia inalterata la Current. Cambia sempre il RowState a *Unchanged*

**UpdateCurrentValues** sovrascrive le DataRowVersion *Current*, ma lascia inalterata la Original. Il RowState cambia nel seguente modo:

- le righe aggiunte hanno *Added*
- le righe con stato Unchanged mantengono *Unchanged* se current è uguale a original, altrimenti viene impostato *Modified*

Per creare una copia completa (schema e dati) di un DataTable si usa il suo metodo *Copy*, utile ad esempio per supportare modifiche che prevedono un abort: un modo semplice è copiare la DataTable e reimpostarla al momento dell'apply. Per copiare solo lo schema ma non i dati si usa invece *Clone*.

*DataView* consente di visualizzare una DataTable impostando inoltre le proprietà *Sort*[14], RowFilter[15], RowStateFilter.

Il *DataSet* è la visuale relazionale dei dati. Contiene un'insieme di DataTable e di *DataRelation* tra varie DataTable e di creare quindi vincoli di integrità referenziale tra le DataTable collegate. Per creare id univoci si utilizza la classe *Guid*, che è una chiave surrogata: non ha senso modificare il valore assunto, ma serve solo a definire l'univocità della chiave e a collegare varie tabelle. Guid consente qiundi di simplificare il join di tabelle nel caso particolare di chiavi multiple. Senza l'uso di Guid, nel contesto di chiavi multiple si richierebbe di andare incontro a problemi di aggiiornamento ricorsivo.

Normalmente è buona pratica inserire versioni fortemente tipizzate degli oggetti DataSet, cioè sostituire espressioni come: footnotesize

```
DataTable companyTable = salesData.Tables["Company"];
```

con una classe che estende DataSet, che contiene inoltre:

```
DataTable companyTable = vendorData.Company;
```

Si può generare automaticamente la versione fortemente tipizzata utilizzando il DataSet Editor per modificare il file XSD che descrive il DataSet.

Il DataRelation consente di ottenere le righe che costituiscono il risultato di uno join. In particolare:

- **join in relazioni (1,N)**: metodo DataRow.*GetChildRows(DataRelation)*
- **join in relazioni (N,1)**: metodo DataRow.*GetParentRow(DataRelation)*

Si può specificare il comportamento in caso di update o delete che coinvolgono vincoli di integrità referenziale, impostando la proprietà DataRelation.*ForeignKeyContraint*:

**Cascade** il delete/update si ripercuote anche nella DataRow referenziata

**None** solleva una *InvalidContraintException*

**SetDefault** imposta al valore di default del tipo della DataRow referenziata (0, "", ...)

**SetNull** imposta a null il valore della DataRow referenziata

Si può serializzare e deserializzare il contenuto di un DataTable o di un DataSet in un file xml (*WriteXml*, *ReadXml*) o in uno stream (BinaryFormatter[16]). Gli spazi nei nomi sono convertiti nei caratteri "`_x0020`". Si può impostare il tipo di serializzazione xml tramite la proprietà DataColumn.*MappingType*: *Attribute*, *Element*, *Hidden* (utili nelle colonne di tipo Expression), *SimpleContent*. Si può inoltre scegliere il formato di esportazione di DataRelation.*Nested*. Nel metodo WriteXml è possibile specificare il *XmlWriteMode.WriteSchema* per memorizzare, oltre ai dati, anche lo schema associato. Viceversa, è possibile esportare solo lo schema tramite *WriteXmlSchema*. Tra i formati disponibili, si può salvare anche nel formato *XmlWriteMode.DiffGram* per salvare anche le versioni associate (Current, Original, ...). Il metodo ReadXml consente di leggere in vari modi:

**Auto** sceglie il miglior metodo

**DiffGram** applica i cambiamenti nel DataSet

**Fragment** analizza il caso di multipli nodi radice

**IgnoreSchema** ignora lo schema eventualmente memorizzato e disponibile

**InferSchema** estende il DataSet corrente con le informazioni dello schema

**InferTypedSchema** prova a determinare lo schema basandosi sui dati

**ReadSchema** legge lo schema. Se una DataTable già esiste viene lanciata un'eccezione

[13] i dati immessi devono soddisfare i vincoli di metadati specificati in DataColumn

[14] prevede una stringa nel formato, ad esempio: "LastName ASC, Salary DESC"

[15] coincide con la clausola WHERE dell'sql, senza il WHERE

[16] necessario impostare anche la proprietà DataSet.*RemotingFormat* a Binary

Bisogna scegliere accuratamente il formato: il formato binario, nonostante abbia dimensioni ridotte, contiene un maggior overhead di 20kb rispetto al formato xml. Quindi, nel caso di piccoli DataSet conviene il formato xml, altrimenti quello binario.

E' possibile combinare i dati di più DataSet tramite il metodo DataTable.*Merge*, che modifica il DataSet correntemente in uso e che prevede un *MissingSchemaAction*:

**Add** aggiunge le necessarie DataTable e DataColumn

**AddWithPrimaryKey** come Add, ma aggiunge anche le PrimaryKey

**Error** errore nel caso in cui è necessario un Add. Quindi se il merge viene applicato a una colonna inesistente non la aggiunge ma solleva un errore

**Ignore** ignora dati che risiedono in DataColumn che non trovano corrispondenza

Quando si utilizza il Merge è necessario che ogni DataTable abbia definita una chiave primaria, altrimenti non si è in grado di identificare le righe e verranno sempre aggiunte

## 5.2 Classi ADO.NET connesse

Le *classi connesse* o *provider class* rappresentano il bridge che consente di trasferire dati tra classi disconnesse e un data store. Il framework .NET contiene i seguenti provider:

**OleDb** accesso generico ai dati verso SQL Server 6.5, SyBase, DB2/400 e Microsoft Access

**Odbc** accesso generico ai dati quando non sono disponibili provider più recenti

**SQL Server** simile a OleDb, ma consente di accedere a SQL Server 7.0 e superiori

**Oracle** accesso per Oracle 8i e superiori

**...** accesso a ulteriori provider specializzati scaricabili da internet, come DB2 e MySql

Per utilizzare le classi connesse è possibile creare esplicitamente le classi specializzate le quali, anche se abilitano codice specializzato, richiedono però una conoscenza esplicita di tali classi da parte dell'utilizzatore. Per ovviare a questo problema si utilizzano le classi concrete *DbProviderFactory* (OracleClientFactory, OdbcFactory, ...), creabili tramite *GetFactory* e che consentono metodi *CreateConnection*, *CreateCommand*, *CreateParameter*, *CreateDataSourceEnumerator*, *CreateDataAdapter*. Ogni classe concreta è implementata come Singleton e l'elenco delle DbProviderFactory disponibili è accessibile tramite *DbProviderFactories.GetFactoryClasses()* e specificabile nelle config, nell'elemento `<system.data> <DbProviderFactories>`. Ogni classe connessa solleva eccezioni specializzate, che derivano da *DbException*.

Una *DbConnection* deve essere creata, aperta e chiusa. Per creare una connessione è necessaria una connection string valida, in cui si può specificare:

**File Name** il percorso fisico di un file che contiene la connection string

**Driver, Provider** il driver ODBC da utilizzare (anche text, excel, ...)

**Server** il nome del server a cui connettersi

**UID** nome utente

**PWD, Password** password utente

**DBQ** il path fisico verso il data source

**AttachDbFilename, Extended Properties, Initial File Name** percorso del file (spesso *.mdf*) nel filesystem locale contenente il database. Per la definizione del path viene supportata la keyword —*DataDirectory*— che punta alla cartella *Data* dell'applicazione. Ottiene o crea il file di log `<database>_log.ldf`.

**Database, Initial Catalog** il database a cui connettersi

**Data Source, addr, address, network address, server** il nome del server a cui connettersi, locale o remoto. Se remoto prevede un indirizzo IP nella forma *IP,PORT*. Per specificare un DBQ si deve anteporre il carattere '.
', nella forma .
*PATH*.

**Connect Timeout** (15) quanti secondi attendere per stabilire la connessione

**Trusted_Connection** se la sicurezza è basata sull'account di dominio dell'utente collegato

**Persist Security Info** se restituire l'intera connection string, compresa quindi delle informazioni di sicurezza

**Integrated Security, trusted_connection** (false) se stabilire una connessione sicura al SQL Server. Valori possibili: true, false e sspi

**MultipleActiveResultSets** (false) abilita MARS: consente di eseguire comandi multipli senza che sia necessario chiudere la connessione

**Asynchronous Processing, async** (false) abilita APM nei metodi di esecuzione delle query (BeginExecuteReader, EndExecuteReader, ...)

**Pooling** (true) abilita il connection pool, che consente di mantenere un pool di connessioni attive in modo da ridurre l'overhead richiesto per creare una connessione e impedire fenomeni di starvation. Risulta particolarmente utile distribuire il pool in un cluster di server, ma in questo caso devono valere le seguenti regole:

1. la connection string deve essere esattamente la stessa tra i vari server
2. l'uid deve essere lo stesso
3. l'id del processo deve essere lo stesso

Il pool viene creato e mantenuto nella stessa macchina che ha creato la DbConnection. Gradualmente viene adattato il numero di connessioni aperte, rilasciando le inattive dopo un tempo casuale compreso tra 4 e 8 minuti

**Min Pool Size** (0) minimo pool da mantenere. Utile inserire un numero basso, come 5, per garantire risposte rapide anche dopo lunghi periodi di inattività

**Max Pool Size** (100) massimo pool da mantenere, la 101esima connessione attende in coda di inserirsi nel pool fino a un timeout stabilito

**Connection Reset** (true) se, al momento della rimozione di una connessione dal pool, inserire informazioni di reset sulla connessione

**Load Balancing Timeout, Connection Lifetime** (0) utile per bilanciare il carico tra un server online e un server appena avviato

**Enlist** se inserire automaticamente la connessione nel contesto di transazione corrente del thread

Tramite `aspnet_regiis.exe` si può cifrare (`-pef`) e decifrare ((`-pdf`)) il contenuto di una connection string che usa le librerie DPAPI o RSA.

Una volta ottenuta la DbConnection si devono impostare le proprietà *CommandType* e *CommandText*, passando eventualmente una lista di *DbParameter* per la stored procedure, le quali richiedono le proprietà *ParameterName* e *Value*. Una volta impostato il comando si può eseguire in tre modi:

**ExecuteNonQuery** non aspetto un risultato (come in un update o delete). Restituisce il numero di tuple condizionate dall'operazione

**ExecuteScalar** mi aspetto un singolo risultato (voglio il numero di dipendenti)

**ExecuteReader** restituisce un *DbDataReader*, che rappresenta un cursore unidirezionale lato server di sola lettura.

ExecuteReader diventa quindi un metodo adatto a popolare dati ad esempio in una GridView, ma non per modificare dati letti e rispedirli al server. Per popolare una GridView si deve prima eseguire il metodo *DataTable.Load*, che richiede un DbDataReader, ed utilizzare la classe disconnesse DataTable per popolare GridView impostando la proprietà DataSource ed eseguendo il DataBind. Il metodo Load prevede una *LoadOption*, come già analizzato:

**OverwriteChanges** come OverwriteRow

**PreserveChanges** come PreserveCurrentValues

**Upsert** come UpdateCurrentValues

*Multiple Active Result Sets MARS* consente di eseguire comandi molteplici sulla stessa connessione, senza che sia necessario chiudere la connessione, consentendo di creare comandi annidati. Non aggiunge potere espressivo[17] ma in alcune situazioni facilita lo sviluppo, anche se peggiora le prestazioni.

Una DbConnection può essere monitorata sottoscrivendo l'evento DbConnection.*InfoMessage*.

Una DbConnection può creare *DbTransaction* tramite DbConnection.*BeginTransaction*, a cui deve seguire un DbTransaction.*Commit* o DbTransaction.*Rollback*.

Per leggere/scrivere dati molto grandi (*Binary Large OBject — BLOB*) è necessario trasferirli in blocchi, in modo da non esaurire la RAM del computer. Per questo, in fase di lettura si può trattare il DbDataReader come un flusso unidirezionale in cui è possibile leggere solo colonna per colonna e scrivere a blocchi il contenuto, ad esempio, in un FileStream. Si può abilitare questo comportamento passando l'enumerato *CommandBehavior.SequentialAccess* a DbCommand.ExecuteReader. Viceversa, in fase di scrittura è possibile scrivere nel database grazie alla seguente sintassi SQL:

1. ottengo il puntatore alla cella da scrivere

   ```
   SELECT TEXTPTR(Immagine) FROM Persona WHERE Matricola = '252762'
   ```

2. utilizzando il puntatore ottenuto, scrivo man mano a blocchi

   ```
   UPDATETEXT Persona.Immagine ImmaginePtr  0 null fiow3ef1ji5oejf
   UPDATETEXT Persona.Immagine ImmaginePtr 32 null sdchduivhuiwhvv
   UPDATETEXT Persona.Immagine ImmaginePtr 64 null qjiorhviuercqoi
   ```

   dove i parametri sono:

   (a) puntatore correntemente in uso. Non c'è quindi bisogno di specificare i vincoli di update, poiché la sola presenza del buffer indica con precisione cosa scrivere

   (b) offset: da dove iniziare la scrittura

   (c) byte da cancellare

   (d) i dati concreti che devono essere inviati

## 5.3 Bridge classi disconnesse – classi connesse

Per copiare i dati da una classe connessa a una classe disconnessa si utilizzano i metodi di *DbDataAdapter* (SqlDataAdapter, OracleDataAdapter, ...) tramite il metodo *Fill*, che richiede che sia stata impostata la proprietà *SelectCommand* a un oggetto *DbCommand* valido, che consente di impostare quale particolare insieme di tuple selezionare per popolare la classe disconnessa.

Viceversa, per copiare dati da una classe disconnessa a una classe connessa si utilizzano la funzione *SqlBulkCopy.WriteToServer*, impostando prima la proprietà SqlBulkCopy.*DestinationTableName*. Per trasferire solo i cambiamenti si usa di nuovo DbDataAdapter, ma stavolta il metodo *Update*, che richiede che siano state impostate le proprietà (in teoria opzionali) *InsertCommand*, *UpdateCommand* e *DeleteCommand*. In realtà questi metodi sono deducibili a partire dalla proprietà (obbligatoria) SelectCommand: infatti dopo la sua esecuzione è possibile ottenere il DbDataReader.*GetSchemaTable()*, che restituisce la DataTable contenente lo schema di colonne previsto e la chiave primaria prevista (una riga per ogni colonna: una metatable). A partire da queste informazione è possibile determinare a quale schema di dati applicare i metodi insert, delete e update senza che sia necessario definirli esplicitamente. Infatti i metodi generati sono definibili come:

```
// INSERT
INSERT INTO %TABLE_NAME%
    [%COLUMN_1%], [%COLUMN_2%]
VALUES
    @%ARG_1%, @%ARG_2%

// UPDATE
UPDATE %TABLE_NAME%
SET
    [%COLUMN_1%] = @%ARG_1%
    [%COLUMN_2%] = @%ARG_2%
WHERE
    (([%COLUMN_PK%] = @%ARG_PK%))

// DELETE
DELETE FROM
    %TABLE_NAME%
WHERE
    (([%COLUMN_PK%] = @%ARG_PK%))
```

[17] con due select annidati in realtà eseguo una join

Questo approccio garantisce migliore sicurezza rispetto a scrivere stored procedure apposite, poiché elimina completamente rischi di attacchi SQL Injection. E' possibile trasferire l'update a blocchi, invece che tupla per tupla utilizzando la proprietà UpdateBatchSize. L'utilizzo complessivo è il seguente:

```
DataSet pubsDataSet = new DataSet("Pubs");

string myProvider = ConfigurationManager.
          ConnectionStrings["PubsData"].ProviderName;
string myConnectionString = ConfigurationManager.
          ConnectionStrings["PubsData"].ConnectionString;
DbProviderFactory fact =
          DbProviderFactories.GetFactory(myProvider);

using(DbConnection conn = fact.CreateConnection()) {
    conn.ConnectionString = myConnectionString;
    conn.Open();

    DbCommand sel = conn.CreateCommand();
    sel.CommandType = CommandType.Text;
    sel.CommantText = "SELECT * FROM Publishers";

    DbDataAdapter da = fact.CreateDataAdapter();
    da.SelectCommand = sel;
    da.UpdateBatchSize = 3; // 3 tuple per volta

    da.Update(pubsDataSet, "publishers"); // ERRORE!

    DbCommandBuilder bldr = fact.CreateCommandBuilder();
    bldr.DataAdapter = da;
    da.InsertCommand = bldr.GetInsertCommand();
    da.UpdateCommand = bldr.GetUpdateCommand();
    da.DeleteCommand = bldr.GetDeleteCommand();

    da.Update(pubsDataSet, "publishers"); // OK!
}
```

Si può utilizzare il DbDataAdapter concreto *OleDbDataAdapter*, che consente di leggere record legacy ADO in un DataSet. Diventa così possibile visualizzare in componenti GUI dati legacy.

## 5.4 Dati in XML

Anche se esistono le classi connesse e disconnesse, il formato XML viene trattato in modo speciale per la sua portabilità. Il W3C ha elaborato una interfaccia standard di programmazione denominata *DOM* (*Document Object Model*), in grado di eseguire accesso casuale ai dati e modifiche di documenti xml. Le funzioni xml derivano dal namespace System.Xml ed estese da System.Data. Esistono diverse classi per gestire documenti xml, ognuna con vari gradi di funzionalità. Per la lettura di un file xml da disco e la creazione della corrispondente versione xml a oggetti esistono le classi:

**Xml[Data** Document] rappresenta la versione ad oggetti di un xml e implementa le specifiche DOM di livello 2. XmlDataDocument è una sua estensione che rappresenta anche dati relazionali. Si possono creare elementi (*CreateElement*), attributi dell'elemento correntemente in uso (CreateAttribute) e aggiungerli all'oggetto (*AppendChild*). Nell'header è necessario inoltre aggiungere un elemento di intestazione, ottenibile tramite *CreateXmlDeclaration*. Si può caricare tale struttura in memoria tramite il metodo XmlDocument.*Load*. Gli elementi e gli attributi sono accessibili in modo programmatico tramite gli iteratori. E' possibile ricercare un *XmlNode* tramite *GetElementByID*, *GetElementByTagName*, *SelectSingleNode* e *SelectNodes*, in cui gli ultimi due consentono di passare una query XPath. Mentre il primo metodo richiede un DTD per determinare quali elementi sono ID, gli altri due non lo richiedono poiché eseguono la query su qualunque tipo di elemento.

**XPathNavigator** consente una navigazione efficiente di un XmlDocument. Dato quindi un XmlDocument, creato tramite il Load, è possibile creare un XPathNavigator tramite XmlDocument.CreateNavigator. La navigazione avviene tramite metodi come *MoveToRoot*, *MoveTo[First—Next]Attribute*, *MoveTo[FirstChild—Next]*, *HasAttributes*, *HasChildren*. Anche se funziona con XmlDocument, dovrebbe se possibile essere utilizzato con XmlPathDocument, specificatamente pensato per XPathNavigator. Il metodo *Select* consente di eseguire unaquery XPath e di restituire un *XPathNodeIterator*, che consente di iterare tra elementi XPathNavigator (e non XmlNode, come in XmlDocument). Il motivo per utilizzare XPathNavigator rispetto a XmlDocument stà nella possibilità di ordinare l'output sulla versione compilata della query XPath. Si può compilare una query XPath tramite *XPathExpression.Compile*, utilizzare XPathExpression.*AddSort* e passare tale oggetto al metodo XPathNavigator.Select.

**XmlReader** unidirezionalmente legge e verifica l'xml, ma non genera la corrispondente versione DOM. Consente di passare un *XmlReaderSettings* per far eseguire la validazione, da passare al metodo *XmlReader.Create*. La validazione avviene al momento del XmlReader.*Load*. Alla fine vuole un *Close*. Diventa la soluzione ideale quando c'è la possibilità che l'informazione che si vuole trovare è verso l'inizio del file xml, in quanto fornisce un accesso forward e privo di cache, opposto a quello random degli altri metodi, che prevede un caching. Prevede metodi di scansione molto simili all'XPathNavigator. XmlReader è una classe astratta: implementazioni concrete sono *XmlTextReader* o *XmlValidatingReader*. Infine, è possibile ottenere la propria implementazione di XmlReader semplicemente estendendo XmlReader

Quando si lavora con oggetti XML risulta quindi più semplice lavorare con i metodi DOM forniti dalla classe XmlDocument, ma ci sono limiti nelle capacità di ricerca che possono implicare un intera scansione dell'albero per raggiungere l'output desiderato. D'altra parte, XPathNavigator è meno intuitivo ma consente query più sofisticate e diventa la soluzione ideale per contesti di utilizzo xml complessi. Viceversa, per la scrittura di un oggetto xml nel corrispondente file xml esistono le classi:

**XmlWriter** scrive l'xml secondo lo standard XML 1.0, ma senza tradurre da una versione DOM. Utilizza XmlConvert per convertire tra i data type CLR e i data type XSD e tradurre spazi in `_x0020_`. Consente di impostare lo stile e altre caratteristiche. Al termine della scrittura segue un *Close* per non perdere dati. Implementazioni specifiche: *XmlTextWriter*, ...

**Xml[Data** Document] consente di scrivere (*Save*) XmlDocument in un file xml. Tipicamente questo avviene dopo un Load, dopo eventuali *RemoveChild* o *InsertAfter*.

# 6 Controlli Web personalizzati

Esistono tre tipi di Web Control personalizzati:

**user control** è un *template control* (.ascx, non .aspx) che fornisce funzionalità extra a controlli indiduali. Deve essere incluso (assieme al code-behind) in ogni progetto che ne fa uso e non può essere quindi globalmente installato. Può essere una composizione di controlli o anche un controllo individuale. Quest'ultimo caso è giustificato dal voler mantenere indipendente il comportamento del control dal suo utilizzo (pattern Bridge)

**custom web control** eredita da un Web Control esistente o ne crea uno ereditando direttamente da WebControl, tipicamente sovrascrivendo inoltre il metodo *Render*. Può essere compilato in un file dll installabile nella GAC

**composite control** custom web control che contiene controlli individuali, utile quando diversi controlli hanno proprietà semantiche in comune. Può essere compilato in un file dll installabile nella GAC

Gli user control ereditano da *UserControl*, che ereditano da *TemplateControl*, che ereditano da Control. Nel file .ascx viene inserita la direttiva *Control* al posto di Page. Si può aggiungere il controllo in due modi:

- aggiungendo la direttiva *Register* e quindi assegnando un *TagPrefix*, un *TagName* e un *Src*
- trascinando il file dal Solution Explorer al design della pagina web. Visual Studio genera automaticamente la direttiva Register richiesta

Le proprietà e gli eventi impostabili vengono esposti (tipicamente avvolti o inoltrati dal web control a cui ci si riferisce). Se l'user control prevede degli stili è necessario prevedere un *controllo Panel radice* che consente di gestire la proprietà Style in tutti i controlli figli aggiunti.

E' possibile creare un *templated user control* che non prevede una GUI, ma consente all'utilizzatore di tale GUI di specificarne una (allo stesso modo di GridView o altri control). Per abilitare questo comportamento bisogna inserire nel ascx un control *PlaceHolder* per stabilire dove apparirà il template. Il code behind dell'uc (user control) definisce una proprietà *ITemplate* che contiene gli attributi *PersistenceMode* e *TemplateContainer*: quest'ultima definisce il contenitore dei valori template ricavabili nella pagina aspx tramite la sintassi:

```
<uc1:ShipperControl Phone="065500445">
  <ShipperTemplate>
    <h1>
      <%# Container.Telefono %>
    </h1>
  </ShipperTemplate>
</uc1:ShipperControl>
```

La proprietà ITemplate deve essere quindi popolata con la classe TemplateContainer dichiarata, e questo avviene nell'init tramite ITemplate.*InstantiateIn*.

Nei casi di *custom control* non esiste invece codice ascx, ma solo cs. Significa che non abbiamo il supporto code behind di Visual Studio ed è necessario far ricorso a metodi come *FindControl*. Inoltre, nel caso di *templated custom control* sono necessarie tre ulteriori operazioni:

- dichiarare nel custom control l'attributo *ParseChildren(true)*
- sovrascrivere il metodo *CreateChildControls* in modo da popolare l'ITemplate tramite InstantiateIn
- sovrascrivere il metodo DataBind in modo da forzare il caricamento prima del DataBind base tramite *EnsureChildControls*

Si noti una cosa: *non ha senso creare un custom control semplicemente per avvolgere proprietà condivise*. Ad esempio, non ha senso creare un controllo AlertButton che utilizza un Button e dichiara il colore rosso. Per questo esistono gli skin.

## 6.1 Aggiungere controlli alla ToolBox

Quando i controlli sono compilati in una dll, possono essere aggiunti alla ToolBox. Si può specificare nella ToolBox:

**Icona del controllo** attributo *ToolboxBitmap* nella definizione del custom control. Deve essere un percorso assoluto, o il namespace dell'immagine bmp (16x16), compilata come Embedded Resource

**Markup iniziale** al momento del primo inserimento, Visual Studio inserisce dei tag di default, impostabili tramite l'attributo *ToolboxData*

**Designer custom** si può realizzare un designer per il controllo creando una classe che eredita da *ControlDesigner* e sovrascrivendo (ad esempio) il metodo ControlDesigner.*GetDesignTimeHtml*. Si può dichiarare il designer realizzato aggiungendo l'attributo *Designer* al custom control

# 7 Gestione dello stato

Le pagine raramente viaggiano da sole, ma hanno bisogno di memorizzare informazioni per personalizzare la pagina. Per questo è necessario una *gestione dello stato*. ASP.NET fornisce due tipi di gestione dello stato:

**lato client** il client comunica al server: "Sono Ricibald, ho 2 libri nella shopping cart"

**lato server** il client comunica al server: "Sono la sessione 432". Il server cerca la sessione 432 e rileva: "La sessione 432 è Ricibald, che ha 2 libri nella shopping cart

Lato client abbiamo una migliore scalabilità: lo stato grava solo sul client, e le richieste possono essere ricevute da server web multipli, i quali altrimenti avrebbero richiesto un bilanciamento intelligente del carico o una gestione centralizzata dello stato. Viceversa, lato server abbiamo maggiore sicurezza delle informazioni dell'utente, che non possono essere manipolate da nessun altro al di fuori del server, e una banda richiesta ridotta, poiché lo stato non deve essere inviato da parte del browser (utile soprattutto nei client mobili, che utilizzano connessioni lente). Inoltre si consideri che qualunque stato memorizzato sul client deve poter essere ripristinabile: i cookie possono essere cancellati, il computer formattato, ...

Solitamente si usa un approccio combinato: il client memorizza la sua sessione, ma non i suoi dati sensibili, mentre il server memorizza i dati sensibili. In questo modo il client non è costretto a identificarsi, mentre la riservatezza dei dati è garantita.

## 7.1 Gestione lato client

La gestione lato client è il modo più scalabile di memorizzare i dati. Esistono differenti tecniche:

**View State** consente di tracciare il valore assunto dai controlli. Per fare questo l'hash dello stato assunto dalla pagina viene memorizzato in un campo nascosto, che viene reinviato tramite HTTP POST a ogni postback. Se il valore hash è troppo lungo per un singolo campo nascosto (a seconda del valore di *MaxPageStateFieldLength*) allora viene splittato tramite il *view state chunking*. Si può cifrare impostando *ViewStateEncryptionMode* al valore *Always*, sia globalmente nel Web.config sia localmente nella Page. Si possono inserire inoltre qualunque valore custom serializzabile nel dictionary Page.*ViewState*. Per accedere ai valori assunti

**Control State** consente di tracciare il valore assunto da uno specifico controllo. A differenza del View State, questo non può essere disabilitato, e risulta utile in controlli in cui sia propedeudico la memorizzazione dello stato per il corretto funzionamento. Per abilitarlo, il controllo deve sovrascrivere OnInit e invocare *RegisterRequiresControlState* e sovrascrivere i metodi *SaveControlState* e *LoadControlState*

**Hidden field** memorizzano dati in un form HTML senza mostrarli all'utente e sono disponibili al submit della form. Non sono però disponibili funzioni importanti come compressione, cifratura, hashing e chunking: conviene quindi aggiungere i valori custom non in hidden field, ma nel view state

**Cookie** memorizzano uno o più valori nel browser dell'utente, i quali vengono reinviati dallo stesso browser ad ogni richiesta verso lo stesso server. I cookie vengono inviati nell'header della risposta, e per questo si può gestire una *HttpCookieCollection* in *Response.Cookies*. I valori HttpCookie.*Value* possono essere persistenti (salvati nel client) o temporanei (eliminati alla fine della sessione) a seconda della presenza del valore di HttpCookie.*Expires*. La visibilità di un cookie può essere limitata a un HttpCookie.*Domain*[18] o a un HttpCookie.*Path*. Si possono impostare anche valori multipli poiché in realtà HttpCookie è a sua volta una dictionary.

**Query string** memorizzano valori tramite una HTTP GET, utili per poter visualizzare i dati inviati al server direttamente nell'URL (memorizzazione nei preferiti, invio in una mail, ...) e per trasferire informazioni di stato tra pagine multiple. Per creare una query string semplicemente bisogna creare l'appropriato URL nella forma, ad esempio:

```
http://www.google.it/search?q=riccardo+pietrucci&sourceid=navclient-ff&ie=UTF-8
```

I valori separati da '&' sono accessibile tramite il dictionary *Request.QueryString*. Attenzione alla lunghezza dell'URL: in un browser massimo 2083 caratteri, in un IM massimo 400, in una mail massimo 70. Bisogna prestare *estrema attenzione alla validazione dell'URL*: può essere facilmente modificato, come visualizzare 100000 risultati nella ricerca, causando attacchi Deny of Service o includere script html come valore. Per evitare questi problemi è necessario utilizzare il Validation Framework e il Server.HtmlEncode.

[18] per ragioni di sicurezza non dovrebbe mai essere impostato "contoso.com": questo potrebbe essere www.contoso.com, intranet.contoso.com, ma anche crack.contoso.com. Impostare invece un full hostname

## 7.2 Gestione lato server

ASP.NET fornisce due modi per condividere informazioni tra pagine web senza inviare dati al client:

**Application state** disponibili in tutte le pagine, ma non memorizzate. Coincide con la dictionary Page.*Application*. Per inizializzare i valori si utilizzano cinque eventi: `Application_Start`, `Application_End`, `Application_Error`, `Session_Start`, `Session_End`. Per implementare gli eventi bisogna aggiungere la Global Application Class *Global.asax*. Bisogna infine stare attenti a problemi di sincronizzazione e racchiudere modifiche concorrenti in blocchi lock/unlock (disponibili nello stesso Application).

**Session state** disponibili da un singolo utente, ma non memorizzate. Coincide con la dictionary Page.*Session*. Consente di mappare l'identificazione del browser alle istanze di sessione nel server e di scambiare dati tra le pagine senza far uso delle query string. Può essere configurata per utilizzare tecniche differenti dai cookie, come le query string, tramite *sessionstate cookieless*. In web.config o nella page si può anche scegliere la modalità di memorizzazione tramite *sessionstate mode*, che può assumere i valori *InProc* (memoria server), *StateServer* (servizio che preserva stato), *SQLServer* (su db), *Custom* e *Off*

I dati memorizzati non sono permanenti e sono perduti a ogni riavvio dell'applicazione, che avviene regolarmente da IIS per migliorare l'affidabilità. Per mantenere tali valori si possono utilizzare i *Profile Property*, che consentono di gestire informazioni utente senza progettare un database apposito. Possono persistere tra processi multipli come nelle Web Farm o nei Web Garden.

# 8 Validazione dell'input e navigazione

## 8.1 Validazione dell'input

Il *Validation Framework* consente validazione sia lato client (maggiori prestazioni) che lato server (maggiore sicurezza). Basta collegare uno o più *controlli validator* a un controllo che accetta input utente. Tutti i controlli ereditano dalla classe *BaseValidator*: qualunque validator necessita che siano impostate le proprietà *ControlToValidate*, *ErrorMessage*, *ToolTip*, *Text*, *Display*. Il controllo *ValidationSummary* consente di mostrare tutti i messaggi di errore dei controlli verificati in un singolo controllo. La lista dei validatori è disponibile tramite *Page.Validators*; il metodo *Validate* imposta il valore di *IsValid* e viene richiamato automaticamente dall'evento *Load*. La classe Control ha un metodo *Focus* che imposta il focus per uno specifico controlli tramite codice JavaScript; il BaseValidator contiene inoltre il metodo *SetFocusOnError* che imposta automaticamente il focus in caso di errore. I controlli che ereditano da BaseValidator sono:

**RequiredFieldValidator** verifica se un campo ha un valore diverso da stringa vuota o da *InitialValue*

**RegularExpressionValidator** verifica se un campo soddisfa una espressione regolare

**CompareValidator** verifica che ControlToValidate è *Operator* ($=$, $\leq$, $<$, *neq*, ...) rispetto a *ControlToCompare* o *ValueToCompare* (costante), entrambi del *Type* specificato

**RangeValidator** verifica i bound *MinimumValue* e *MaximumValue* del controllo del *Type* specificato

**CustomValidator** consente di specificare codice di validation lato client (*ClientFunctionName*) tramite JavaScript e codice di validation lato server (*ServerValidate*) tramite un qualunque linguaggio .NET. Entrambi i metodi devono avere la signature:

```
protected void functionName(object source,
          ServerValidateEventArgs args)
```

Uno dei benefici della validazione lato client è il fatto che il post della pagina non viene eseguito finquando tutte le validazioni non sono verificate positivamente. Questo può essere un problema nel caso si prevedano però pulsanti speciali come Cancel o Help. Si può fare in modo che alcuni pulsanti ignorino la validazione impostando la proprietà *CausesValidation* a false. Inoltre, un pulsante potrebbe verificare non tutti i controlli, ma solo quelli di una specifica sezione. Per questo è possibile identificare nelle validation e nei controlli che causano postback il *ValidationGroup* di appartenenza, e ottenibili tramite Page.*GetValidators*, che consente di specificare il validation group.

## 8.2 Site navigation

I controlli che eseguono una PostBack consentono di trasferire facilmente lo stato, ma esistono contesti in cui è necessario memorizzare dati durante la navigazione. Se le soluzioni fornite da Wizard o FormView, che utilizzano il controllo Pager, non sono sufficienti, esistono cinque possibili soluzioni:

**Codice lato client o markup** consente di richiedere la navigazione verso una pagina, o tramite un HyperLink o tramite codice JavaScript

**PostBack tra pagine** configura un controllo per eseguire una PostBack tra differenti pagine web, in modo che i dati della prima pagina sono accessibili dalla seconda tramite Page.*PreviousPage*. Il valore del controllo da ottenere è accessibile utilizzando FindControl, ma non è conveniente poiché bisogna conoscere l'id del controllo. Si può invece impostare nella seconda pagina la direttiva *PreviousPageType* per avere un accesso tipizzato alle proprietà della prima pagina

**Trasferimento lato client** viene indicato il redirect nella risposta html, tramite Response.*Redirect*. Per garantire che nessun dato venga trasferito fino all'esecuzione del redirect si imposta a true il Response.*BufferOutput*. Con il redirect la proprietà PreviousPage non viene popolata ed è necessario utilizzare i metodi tradizionali per trasferire dati, come cookie, session, QueryString

**Trasferimento lato server** sposta una pagina web differente utilizzando il metodo *HttpUtility.Transfer*. La proprietà Page.Server incapsula un'istanza di HttpUtility, e si può quindi utilizzare il metodo Page.Server.Transfer. Il metodo accetta un booleano *preserveForm* che consente di preservare i valori passati nella QueryString (conveniente inserire false)

**Controlli specializzati** utilizzare controlli specializzati alla navigazione, come Menu, TreeView e *SiteMapPath* che possono prendere come input un SiteMapDataSource (file xml `Web.sitemap`)