1. 读取表中元素提交的数据

表中数据<input type=”text” name=”input\_name”>

服务器端读取context.Request[“input\_name”] or Request[“input\_name”]

1. 响应请求，向客户端输入内容

context.Response.Write(“Hello”) or context.Response.Write(“<input type…..>”)

1. 页面响应问题

当你在Example.html里提交一个这样的表单<form action=”Handler.ashx”>

得到的响应地址是localhost:port/Handler.ashx?somedata…

1. @操作符加在C# string前的作用

例如：@"c:\Docs\Source\a.txt" // rather than "c:\\Docs\\Source\\a.txt"

@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.

1. Response.ContentType

即使在程序中将Response.ContentType=”text/html”，响应的URL仍然是localhost:port/Handler.ashx

1. 使用File类读写html文件

string filePath = context.Server.MapPath("HtmlPage.html");  
string content = System.IO.File.ReadAllText(filePath);  
context.Response.Write(content);

The [MapPath](https://msdn.microsoft.com/en-us/library/ms524632(v=vs.90).aspx) method maps the specified relative or virtual path to the corresponding physical directory on the server.

1. 占位符@

在html中使用占位符

<form action="Handler.ashx">  
        Name:<input type="text" name="text1" value=""/><input type="submit" value="submit" />  
        <input type="hidden" name="isPostBack" value="true"/>  
        Name:<input type="text" name="text1" value="@value"/><input type="submit" value="submit" />  
    </form>  
    @msg

在服务器端替换占位符的值

string filePath = context.Server.MapPath("HtmlPage.html");

string content = System.IO.File.ReadAllText(filePath);

content=content.Replace("@value",name);

content=content.Replace("@msg",msg)；

1. 内存数据的自动增长

在html中<form action="Handler.ashx" method="post">

<input type="hidden" name="isPostBack" value="true" />

Value:<input type="text" name="text1" value="@value" /><input type="submit" value="Increse" />

</form>

服务器端代码

public void ProcessRequest (HttpContext context) {

string value = context.Request["text1"];

string isPostBack = context.Request["isPostBack"];

context.Response.ContentType = "text/html";

if (isPostBack=="true")

{

int temp = Convert.ToInt32(value);

temp++;

value = temp.ToString();

}

else

{

value = "0";

}

string filePath = context.Server.MapPath("HtmlPage.html");

string content = System.IO.File.ReadAllText(filePath);

content = content.Replace("@value", value);

context.Response.Write(content);

}

1. View State

<https://msdn.microsoft.com/en-us/library/ms972976.aspx>

The Role of View State

View state's purpose in life is simple: it's there to persist state across postbacks. (For an ASP.NET Web page, its state is the property values of the controls that make up its control hierarchy.) This begs the question, "What sort of state needs to be persisted?" To answer that question, let's start by looking at what state *doesn't* need to be persisted across postbacks. Recall that in the instantiation stage of the page life cycle, the control hierarchy is created and those properties that are specified in the declarative syntax are assigned. Since these declarative properties are automatically reassigned on each postback when the control hierarchy is constructed, there's no need to store these property values in the view state.

For example, imagine we have a Label Web control in the HTML portion with the following declarative syntax:

<asp:Label runat="server" Font-Name="Verdana"

Text="Hello, World!"></asp:Label>

When the control hierarchy is built in the instantiation stage, the Label's Text property will be set to "Hello, World!" and its Font property will have its Name property set to Verdana. Since these properties will be set each and every page visit during the instantiation stage, there's no need to persist this information in the view state.

What needs to be stored in the view state is any programmatic *changes* to the page's state. For example, suppose that in addition to this Label Web control, the page also contained two Button Web controls, a Change Message Button and an Empty Postback button. The Change Message Button has a Click event handler that assigns the Label's Text property to "Goodbye, Everyone!"; the Empty Postback Button just causes a postback, but doesn't execute any code. The change to the Label's Text property in the Change Message Button would need to be saved in the view state. To see how and when this change would be made, let's walk through a quick example. Assuming that the HTML portion of the page contains the following markup:

<asp:Label runat="server" ID="lblMessage"

Font-Name="Verdana" Text="Hello, World!"></asp:Label>

<br />

<asp:Button runat="server"

Text="Change Message" ID="btnSubmit"></asp:Button>

<br />

<asp:Button runat="server" Text="Empty Postback"></asp:Button>

And the code-behind class contains the following event handler for the Button's Click event:

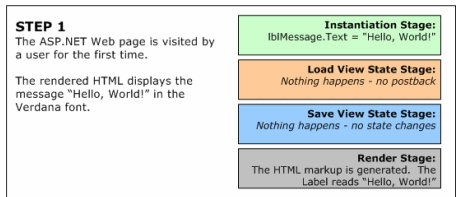
private void btnSubmit\_Click(object sender, EventArgs e)

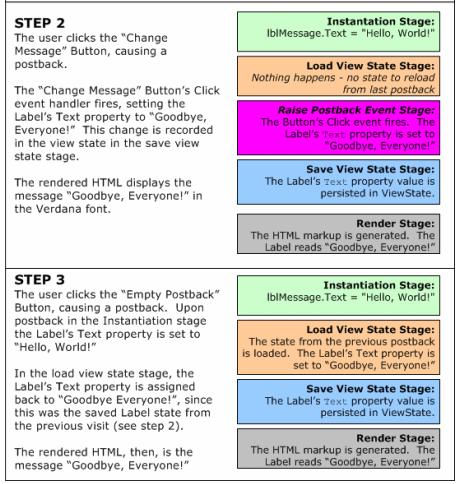
{

lblMessage.Text = "Goodbye, Everyone!";

}

Figure 4 illustrates the sequence of events that transpire, highlighting why the change to the Label's Text property needs to be stored in the view state.





To understand why saving the Label's changed Text property in the view state is vital, consider what would happen if this information were not persisted in view state. That is, imagine that in step 2's save view state stage, no view state information was persisted. If this were the case, then in step 3 the Label's Text property would be assigned to "Hello, World!" in the instantiation stage, but would not be reassigned to "Goodbye, Everyone!" in the load view state stage. Therefore, from the end user's perspective, the Label's Text property would be "Goodbye, Everyone!" in step 2, but would seemingly be reset to its original value ("Hello, World!") in step 3, after clicking the Empty Postback button.

## View State and Dynamically Added Controls

Since all ASP.NET server controls contain a collection of child controls exposed through the Controls property, controls can be dynamically added to the control hierarchy by appending new controls to a server control's Controls collection. A thorough discussion of dynamic controls is a bit beyond the scope of this article, so we won't cover that topic in detail here; instead, we'll focus on how to manage view state for controls that are added dynamically. (For a more detailed lesson on using dynamic controls, check out [Dynamic Controls in ASP.NET](http://aspnet.4guysfromrolla.com/articles/081402-1.aspx) and [Working with Dynamically Created Controls](http://aspnet.4guysfromrolla.com/articles/082102-1.aspx).)

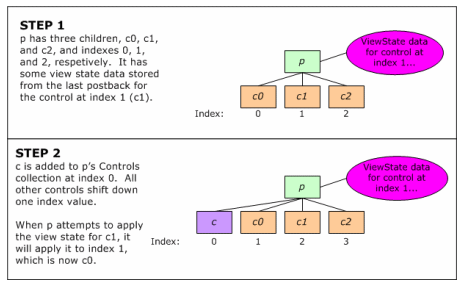
Recall that in the page life cycle, the control hierarchy is created and the declarative properties are set in the instantiation stage. Later, in the load view state stage, the state that had been altered in the prior page visit is restored. Thinking a bit about this, three things become clear when working with dynamic controls:

1. Since the view state only persists changed control state across postbacks, and not the actual controls themselves, dynamically added controls must be added to the ASP.NET Web page, on both the initial visit as well as all subsequent postbacks.
2. Dynamic controls are added to the control hierarchy in the code-behind class, and therefore are added at some point after the instantiation stage.
3. The view state for these dynamically added controls is automatically saved in the save view state stage. (What happens on postback if the dynamic controls have not yet been added by the time the load view state stage rolls, however?)

So, dynamically added controls must be programmatically added to the Web page on each and every page visit. The best time to add these controls is during the initialization stage of the page life cycle, which occurs before the load view state stage. That is, we want to have the control hierarchy complete before the load view state stage arrives. For this reason, it is best to create an event handler for the Page class's Init event in your code-behind class, and add your dynamic controls there.

**Note**   You may be able to get away with loading your controls in the Page\_Load event handler and maintaining the view state properly. It all depends on whether or not you are setting any properties of the dynamically loaded controls programmatically and, if so, when you're doing it relative to theControls.Add(**dynamicControl**) line. A thorough discussion of this is a bit beyond the scope of this article, but the reason it may work is because the Controlsproperty's Add() method recursively loads the parent's view state into its children, even though the load view state stage has passed.

When adding a dynamic control c to some parent control p based on some condition (that is, when not loading them on each and every page visit), you need to make sure that you add c to the end of p's Controls collection. The reason is because the view state for p contains the view state for p's children as well, and, as we'll discuss in the "Parsing the View State" section, p's view state specifies the view state for its children by index. (Figure 5 illustrates how inserting a dynamic control somewhere other than the end of the Controls collection can cause a corrupted view state.)



## The ViewState Property

Each control is responsible for storing its own state, which is accomplished by adding its changed state to its ViewState property. The ViewState property is defined in theSystem.Web.UI.Control class, meaning that all ASP.NET server controls have this property available. (When talking about view state in general I'll use lower case letters with a space between view and state; when discussing the ViewState property, I'll use the correct casing and code-formatted text.)

If you examine the simple properties of any ASP.NET server control you'll see that the properties read and write directly to the view state. (You can view the decompiled source code for a .NET assembly by using a tool like [Reflector](http://www.aisto.com/roeder/dotnet/).) For example, consider the HyperLink Web control's NavigateUrl property. The code for this property looks like so:

public string NavigateUrl

{

get

{

string text = (string) ViewState["NavigateUrl"];

if (text != null)

return text;

else

return string.Empty;

}

set

{

ViewState["NavigateUrl"] = value;

}

}

As this code sample illustrates, whenever a control's property is read, the control's ViewState is consulted. If there is not an entry in the ViewState, then the default value for the property is returned. When the property is assigned, the assigned value is written directly to the ViewState.

**Note**   All Web controls use the above pattern for simple properties. Simple properties are those that are scalar values, like strings, integers, Booleans, and so on. Complex properties, such as the Label's **Font** property, which might be classes themselves, use a different approach. Consult the book [Developing Microsoft ASP.NET Server Controls and Components](http://www.4guysfromrolla.com/ASPScripts/Goto.asp?ID=170) for more information on state maintenance techniques for ASP.NET server controls.

The ViewState property is of type System.Web.UI.StateBag. The StateBag class provides a means to store name and value pairs, using aSystem.Collections.Specialized.HybridDictionary behind the scenes. As the NavigateUrl property syntax illustrates, items can be added to and queried from the StateBagusing the same syntax you could use to access items from a [Hashtable](https://msdn.microsoft.com/en-us/library/aa289149.aspx).

## Timing the Tracking of View State

Recall that earlier I said the view state only stores state that needs to be persisted across postbacks. One bit of state that does not need to be persisted across postbacks is the control's properties specified in the declarative syntax, since they are automatically reinstated in the page's instantiation stage. For example, if we have a HyperLink Web control on an ASP.NET Web page and declaratively set the NavigateUrl property to [http://www.ScottOnWriting.NET](http://www.scottonwriting.net/) then this information doesn't need to be stored in the view state.

Seeing the HyperLink control's NavigateUrl property's code, however, it looks as if the control's ViewState is written to whenever the property value is set. In the instantiation stage, therefore, where we'd have something like HyperLink1.NavigateUrl = http://www.ScottOnWriting.NET;, it would only make sense that this information would be stored in the view state.

Regardless of what might seem apparent, this is not the case. The reason is because the StateBag class only tracks changes to its members after its TrackViewState() method has been invoked. That is, if you have a StateBag, any and all additions or modifications that are made before TrackViewState() is made will not be saved when the SaveViewState()method is invoked. The TrackViewState() method is called at the end of the initialization stage, which happens after the instantiation stage. Therefore, the initial property assignments in the instantiation stage—while written to the ViewState in the properties' set accessors—are not persisted during the SaveViewState() method call in the save view state stage, because the TrackViewState() method has yet to be invoked.

**Note**   The reason the StateBag has the TrackViewState() method is to keep the view state as trimmed down as possible. Again, we don't want to store the initial property values in the view state, as they don't need to be persisted across postbacks. Therefore, the TrackViewState() method allows the state management to begin after the instantiation and initialization stages.

1. ASP.NET Lifecycle

