

ASP.NET Page Life Cycle Overview

When an ASP.NET page runs, the page goes through a life cycle in which it performs a series of processing steps. These include initialization, instantiating controls, restoring and maintaining state, running event handler code, and rendering. It is important for you to understand the page life cycle so that you can write code at the appropriate life-cycle stage for the effect you intend.

If you develop custom controls, you must be familiar with the page life cycle in order to correctly initialize controls, populate control properties with view-state data, and run control behavior code. The life cycle of a control is based on the page life cycle, and the page raises many of the events that you need to handle in a custom control.

This topic contains the following sections:

- [General Page Life-cycle Stages](#)
- [Life-cycle Events](#)
- [Additional Page Life Cycle Considerations](#)
- [Catch-Up Events for Added Controls](#)
- [Data Binding Events for Data-Bound Controls](#)
- [Login Control Events](#)

General Page Life-Cycle Stages

In general terms, the page goes through the stages outlined in the following table. In addition to the page life-cycle stages, there are application stages that occur before and after a request but are not specific to a page. For more information, see [Introduction to the ASP.NET Application Life Cycle](#) and [ASP.NET Application Life Cycle Overview for IIS 7.0](#).

Some parts of the life cycle occur only when a page is processed as a postback. For postbacks, the page life cycle is the same during a partial-page postback (as when you use an [UpdatePanel](#) control) as it is during a full-page postback.

Stage	Description
Page request	The page request occurs before the page life cycle begins. When the page is requested by a user, ASP.NET determines whether the page needs to be parsed and compiled (therefore beginning the life of a page), or whether a cached version of the page can be sent in response without running the page.
Start	In the start stage, page properties such as Request and Response are set. At this stage, the page also determines whether the request is a postback or a new request and sets the IsPostBack property. The page also sets the UICulture property.
Initialization	During page initialization, controls on the page are available and each control's UniqueID property is set. A master page and themes are also applied to the page if applicable. If the current request is a postback, the postback data has not yet been loaded and control property values have not been restored to the values from view state.
Load	During load, if the current request is a postback, control properties are loaded with information recovered from view state and control state.
Postback event handling	If the request is a postback, control event handlers are called. After that, the Validate method of all validator controls is called, which sets the IsValid property of individual validator controls and of the page. (There is an exception to this sequence: the handler for the event that caused validation is called after validation.)
Rendering	Before rendering, view state is saved for the page and all controls. During the rendering stage, the page calls the Render method for each control, providing a text writer that writes its output to the OutputStream object of the page's Response property.
Unload	The Unload event is raised after the page has been fully rendered, sent to the client, and is ready to be discarded. At this point, page properties such as Response and Request are unloaded and cleanup is performed.



Life-Cycle Events

Within each stage of the life cycle of a page, the page raises events that you can handle to run your own code. For control events, you bind the event handler to the event, either declaratively using attributes such as **onclick**, or in code.

Pages also support automatic event wire-up, meaning that ASP.NET looks for methods with particular names and automatically runs those methods when certain events are raised. If the **AutoEventWireup** attribute of the [@ Page](#) directive is set to **true**, page events are automatically bound to methods that use the naming convention of **Page_event**, such as **Page_Load** and **Page_Init**. For more information on automatic event wire-up, see [ASP.NET Web Forms Server Control Event Model](#).

The following table lists the page life-cycle events that you will use most frequently. There are more events than those listed; however, they are not used for most page-processing scenarios. Instead, they are primarily used by server controls on the ASP.NET Web page to initialize and render themselves. If you want to write custom ASP.NET server controls, you need to understand more about these events. For information about creating custom controls, see [Developing Custom ASP.NET Server Controls](#).

Page Event	Typical Use

Preinit	<p>Raised after the start stage is complete and before the initialization stage begins.</p> <p>Use this event for the following:</p> <ul style="list-style-type: none">• Check the IsPostBack property to determine whether this is the first time the page is being processed. The IsCallback and IsCrossPagePostBack properties have also been set at this time.• Create or re-create dynamic controls.• Set a master page dynamically.• Set the Theme property dynamically.• Read or set profile property values. <div><div> Note</div><div>If the request is a postback, the values of the controls have not yet been restored from view state. If you set a control property at this stage, its value might be overwritten in the next event.</div></div>
Init	<p>Raised after all controls have been initialized and any skin settings have been applied. The Init event of individual controls occurs before the Init event of the page.</p> <p>Use this event to read or initialize control properties.</p>
InitComplete	<p>Raised at the end of the page's initialization stage. Only one operation takes place between the Init and InitComplete events: tracking of view state changes is turned on. View state tracking enables controls to persist any values that are programmatically added to the ViewState collection. Until view state tracking is turned on, any values added to view state are lost across postbacks. Controls typically turn on view state tracking immediately after they raise their Init event.</p> <p>Use this event to make changes to view state that you want to make sure are persisted after the next postback.</p>
PreLoad	<p>Raised after the page loads view state for itself and all controls, and after it processes postback data that is included with the Request instance.</p>
Load	<p>The Page object calls the OnLoad method on the Page object, and then recursively does the same for each child control until the page and all controls are loaded. The Load event of individual controls occurs after the Load event of the page.</p> <p>Use the OnLoad event method to set properties in controls and to establish database connections.</p>
Control events	<p>Use these events to handle specific control events, such as a Button control's Click event or a TextBox control's TextChanged event.</p> <div><div> Note</div><div>In a postback request, if the page contains validator controls, check the IsValid property of the Page and of individual validation controls before performing any processing.</div></div>
LoadComplete	<p>Raised at the end of the event-handling stage.</p> <p>Use this event for tasks that require that all other controls on the page be loaded.</p>
PreRender	<p>Raised after the Page object has created all controls that are required in order to render the page, including child controls of composite controls. (To do this, the Page object calls EnsureChildControls for each control and for the page.)</p> <p>The Page object raises the PreRender event on the Page object, and then recursively does the same for each child control. The PreRender event of individual controls occurs after the PreRender event of the page.</p> <p>Use the event to make final changes to the contents of the page or its controls before the rendering stage begins.</p>
PreRenderComplete	<p>Raised after each data bound control whose DataSourceID property is set calls its DataBind method. For more information, see Data Binding Events for Data-Bound Controls later in this topic.</p>
SaveStateComplete	<p>Raised after view state and control state have been saved for the page and for all controls. Any changes to the page or controls at this point affect rendering, but the changes will not be retrieved on the next postback.</p>
Render	<p>This is not an event; instead, at this stage of processing, the Page object calls this method on each control. All ASP.NET Web server controls have a Render method that writes out the control's markup to send to the browser.</p> <p>If you create a custom control, you typically override this method to output the control's markup. However, if your custom control incorporates only standard ASP.NET Web server controls and no custom markup, you do not need to override the Render method. For more information, see Developing Custom ASP.NET Server Controls.</p> <p>A user control (an .ascx file) automatically incorporates rendering, so you do not need to explicitly render the control in code.</p>
Unload	<p>Raised for each control and then for the page.</p> <p>In controls, use this event to do final cleanup for specific controls, such as closing control-specific database connections.</p> <p>For the page itself, use this event to do final cleanup work, such as closing open files and database connections, or finishing up logging or other request-specific tasks.</p>

Note

During the unload stage, the page and its controls have been rendered, so you cannot make further changes to the response stream. If you attempt to call a method such as the **Response.Write** method, the page will throw an exception.

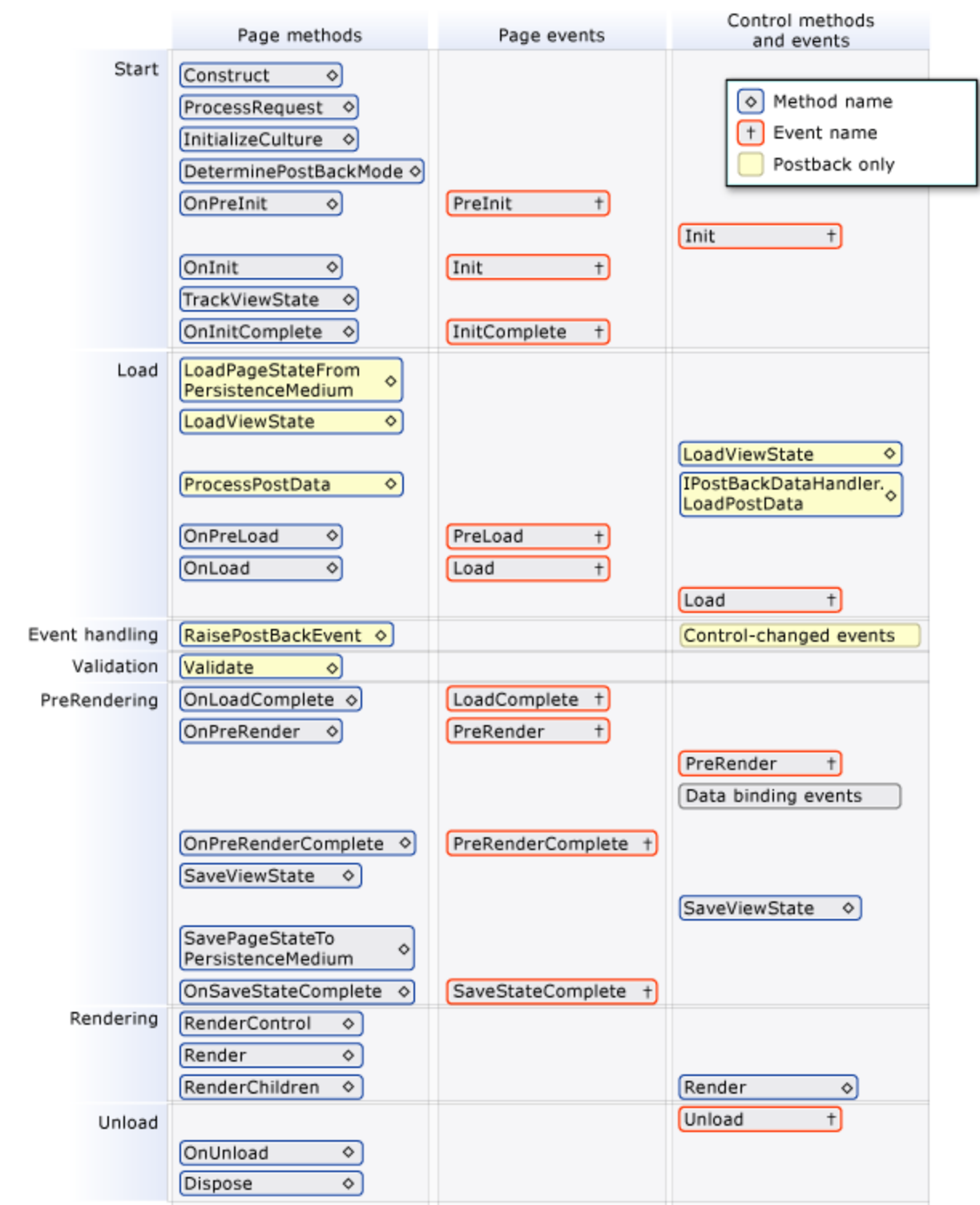
Additional Page Life Cycle Considerations

Individual ASP.NET server controls have their own life cycle that is similar to the page life cycle. For example, a control's [Init](#) and [Load](#) events occur during the corresponding page events.

Although both [Init](#) and [Load](#) recursively occur on each control, they happen in reverse order. The [Init](#) event (and also the [Unload](#) event) for each child control occur before the corresponding event is raised for its container (bottom-up). However the [Load](#) event for a container occurs before the [Load](#) events for its child controls (top-down). Master pages behave like child controls on a page: the master page [Init](#) event occurs before the page [Init](#) and [Load](#) events, and the master page [Load](#) event occurs after the page [Init](#) and [Load](#) events.

When you create a class that inherits from the [Page](#) class, in addition to handling events raised by the page, you can override methods from the page's base class. For example, you can override the page's [InitializeCulture](#) method to dynamically set culture information. Note that when an event handler is created using the **Page_event** syntax, the base implementation is implicitly called and therefore you do not need to call it in your method. For example, the base page class's [OnLoad](#) method is always called, whether you create a **Page_Load** method or not. However, if you override the page [OnLoad](#) method with the **override** keyword (**Overrides** in Visual Basic), you must explicitly call the base method. For example, if you override the [OnLoad](#) method on the page, you must call **base.Load** (**MyBase.Load** in Visual Basic) in order for the base implementation to be run.

The following illustration shows some of the most important methods of the [Page](#) class that you can override in order to add code that executes at specific points in the page life cycle. (For a complete list of page methods and events, see the [Page](#) class.) The illustration also shows how these methods relate to page events and to control events. The sequence of methods and events in the illustration is from top to bottom, and within each row from left to right.



Catch-Up Events for Added Controls

If controls are created dynamically at run time or declaratively within templates of data-bound controls, their events are initially not synchronized with those of other controls on the page. For example, for a control that is added at run time, the [Init](#) and [Load](#) events might occur much later in the page life cycle than the same events for controls created declaratively. Therefore, from the time that they are instantiated, dynamically added controls and controls in templates raise their events one after the other until they have caught up to the event during which it was added to the [Controls](#) collection.

Data Binding Events for Data-Bound Controls

To help you understand the relationship between the page life cycle and data binding events, the following table lists data-related events in data-bound controls such as the [GridView](#), [DetailsView](#), and [FormView](#) controls.

Control Event	Typical Use
DataBinding	<p>Raised after the control's PreRender event, which occurs after the page's PreRender event. (This applies to controls whose DataSourceID property is set declaratively. Otherwise the event happens when you call the control's DataBind method.)</p> <p>This event marks the beginning of the process that binds the control to the data. Use this event to manually open database connections, if required, and to set parameter values dynamically before a query is run.</p>
RowCreated (GridView only) or ItemCreated (DataList , DetailsView , SiteMapPath , DataGrid , FormView , Repeater , and ListView controls)	<p>Raised after the control's DataBinding event.</p> <p>Use this event to manipulate content that is not dependent on data binding. For example, at run time, you might programmatically add formatting to a header or footer row in a GridView control.</p>
RowDataBound (GridView only) or ItemDataBound (DataList , SiteMapPath , DataGrid , Repeater , and ListView controls)	<p>Raised after the control's RowCreated or ItemCreated event.</p> <p>When this event occurs, data is available in the row or item, so you can format data or set the FilterExpression property on child data source controls in order to display related data within the row or item.</p>
DataBound	<p>Raised at the end of data-binding operations in a data-bound control. In a GridView control, data binding is complete for all rows and any child controls.</p> <p>Use this event to format data-bound content or to initiate data binding in other controls that depend on values from the current control's content. (For more information, see Catch-Up Events for Added Controls earlier in this topic.)</p>

Nested Data-Bound Controls

If a child control has been data bound, but its container control has not yet been data bound, the data in the child control and the data in its container control can be out of sync. This is true particularly if the data in the child control performs processing based on a data-bound value in the container control.

For example, suppose you have a [GridView](#) control that displays a company record in each row, and it displays a list of the company officers in a [ListBox](#) control. To fill the list of officers, you would bind the [ListBox](#) control to a data source control (such as [SqlDataSource](#)) that retrieves the company officer data using the company ID in a query.

If the [ListBox](#) control's data-binding properties, such as [DataSourceID](#) and [DataMember](#), are set declaratively, the [ListBox](#) control will try to bind to its data source during the containing row's [DataBinding](#) event. However, the CompanyID field of the row does not contain a value until the [GridView](#) control's [RowDataBound](#) event occurs. In this case, the child control (the [ListBox](#) control) is bound before the containing control (the [GridView](#) control) is bound, so their data-binding stages are out of sync.

To avoid this condition, put the data source control for the [ListBox](#) control in the same template item as the [ListBox](#) control itself, and do not set the data binding properties of the [ListBox](#) declaratively. Instead, set them programmatically at run time during the [RowDataBound](#) event, so that the [ListBox](#) control does not bind to its data until the CompanyID information is available.

For more information, see [Binding to Data Using a Data Source Control](#).

Login Control Events

The [Login](#) control can use settings in the Web.config file to manage membership authentication automatically. However, if your application requires you to customize how the control works, or if you want to understand how [Login](#) control events relate to the page life cycle, you can use the events listed in the following table.

Control Event	Typical Use
LoggingIn	<p>Raised during a postback, after the page's LoadComplete event has occurred. This event marks the beginning of the login process.</p> <p>Use this event for tasks that must occur prior to beginning the authentication process.</p>
Authenticate	<p>Raised after the LoggingIn event.</p> <p>Use this event to override or enhance the default authentication behavior of a Login control.</p>
LoggedIn	<p>Raised after the user name and password have been authenticated.</p> <p>Use this event to redirect to another page or to dynamically set the text in the control. This event does not occur if there is an error or if authentication fails.</p>
LoginError	<p>Raised if authentication was not successful.</p> <p>Use this event to set text in the control that explains the problem or to direct the user to a different page.</p>

See Also

Reference
[Validating User Input in ASP.NET Web Pages](#)

[ASP.NET Login Controls Overview](#)

Concepts

[ASP.NET Web Forms Server Control Event Model](#)

[Page and Application Context in ASP.NET Web Applications](#)

[ASP.NET View State Overview](#)

Other Resources

[Developing Custom ASP.NET Server Controls](#)

[ASP.NET Page Syntax](#)

[Server Event Handling in ASP.NET Web Forms Pages](#)

[Binding to Data Using a Data Source Control](#)