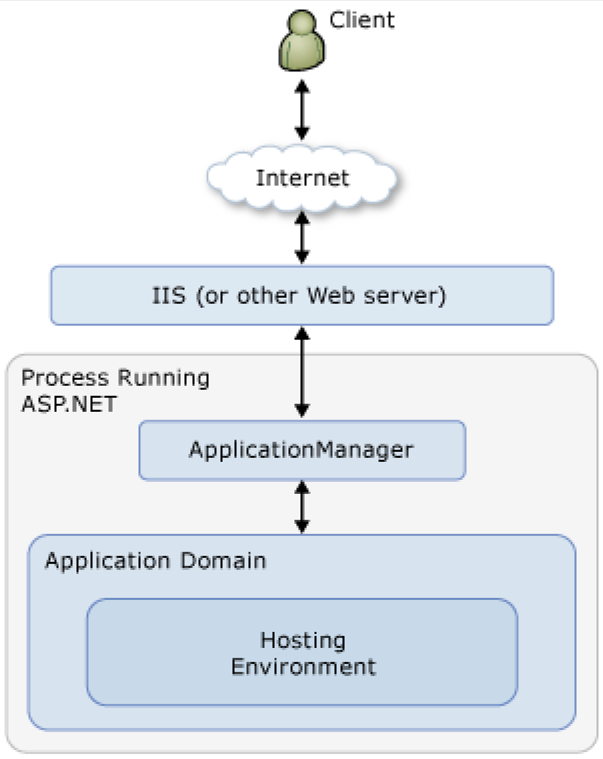
ASP.NET
ASP.NET Application Life Cycle Overview

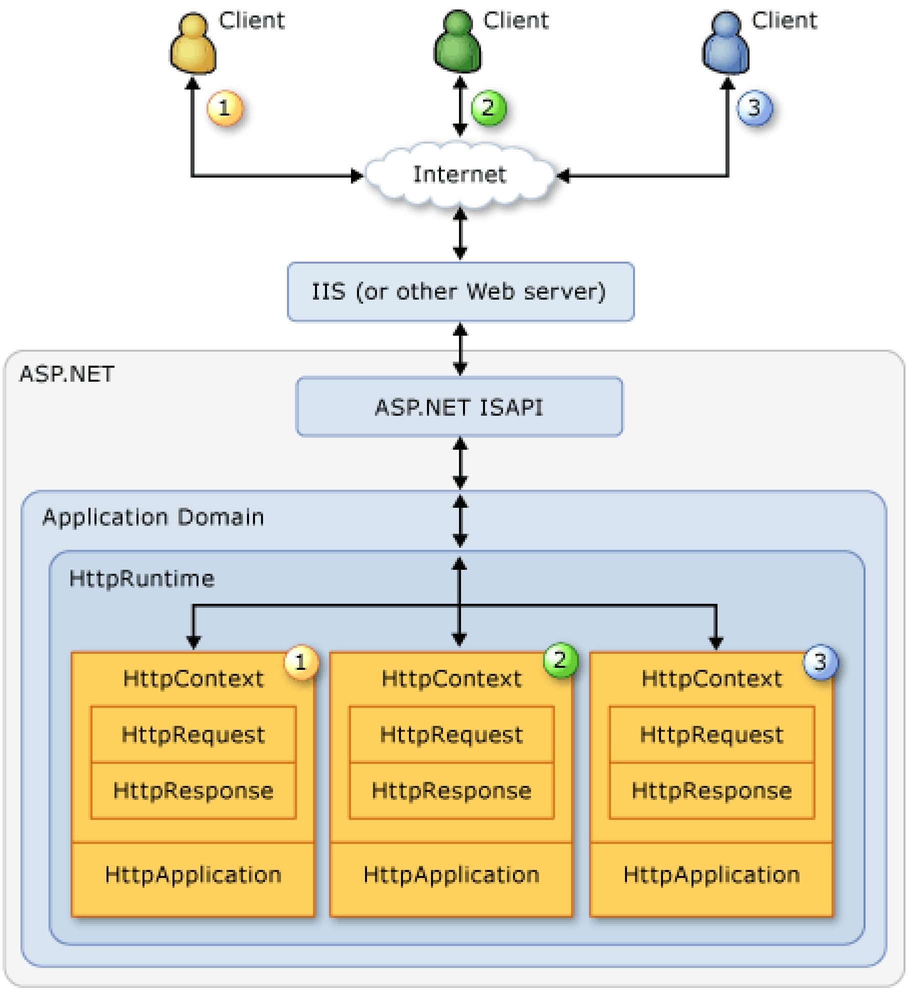
This topic outlines the application life cycle, listing important life-cycle events and describing how code that you write can fit into the application life cycle. Within ASP.NET, several processing steps must occur for an ASP.NET application to be initialized and process requests. Additionally, ASP.NET is only one piece of the Web server architecture that services requests made by browsers. It is important for you to understand the application life cycle so that you can write code at the appropriate life cycle stage for the effect you intend.

Application Life Cycle in General

The following table describes the stages of the ASP.NET application life cycle.

Stage	**Description**
User requests an application resource from the Web server.	The life cycle of an ASP.NET application starts with a request sent by a browser to the Web server (for ASP.NET applications, typically IIS). ASP.NET is an ISAPI extension under the Web server. When a Web server receives a request, it examines the file name extension of the requested file, determines which ISAPI extension should handle the request, and then passes the request to the appropriate ISAPI extension. ASP.NET handles file name extensions that have been mapped to it, such as .aspx, .ascx, .ashx, and .asmx. **Note** If a file name extension has not been mapped to ASP.NET, then ASP.NET will not receive the request. This is important to understand for applications that use ASP.NET authentication. For example, because .htm files are typically not mapped to ASP.NET, ASP.NET will not perform authentication or authorization checks on requests for .htm files. Therefore, even if a file contains only static content, if you want ASP.NET to check authentication, create the file using a file name extension mapped to ASP.NET, such as .aspx. **Note** If you create a custom handler to service a particular file name extension, you must map the extension to ASP.NET in IIS and also register the handler in your application's Web.config file. For more information, see Introduction to HTTP Handlers.
ASP.NET receives the first request for the application.	When ASP.NET receives the first request for any resource in an application, a class named ApplicationManager creates an application domain. Application domains provide isolation between applications for global variables and allow each application to be unloaded separately. Within an application domain, an instance of the class named HostingEnvironment is created, which provides access to information about the application such as the name of the folder where the application is stored. The following diagram illustrates this relationship:

	 Client Internet IIS (or other Web server) Process Running ASP.NET ApplicationManager Application Domain Hosting Environment ASP.NET also compiles the top-level items in the application if required, including application code in the App_Code folder. For more information, see "Compilation Life Cycle" later in this topic.
ASP.NET core objects are created for each request.	After the application domain has been created and the **HostingEnvironment** object instantiated, ASP.NET creates and initializes core objects such as HttpContext, HttpRequest, and HttpResponse. The **HttpContext** class contains objects that are specific to the current application request, such as the **HttpRequest** and **HttpResponse** objects. The **HttpRequest** object contains information about the current request, including cookies and browser information. The **HttpResponse** object contains the response that is sent to the client, including all rendered output and cookies.
An HttpApplication object is assigned to the request	After all core application objects have been initialized, the application is started by creating an instance of the **HttpApplication** class. If the application has a Global.asax file, ASP.NET instead creates an instance of the Global.asax class that is derived from the **HttpApplication** class and uses the derived class to represent the application. **Note** The first time an ASP.NET page or process is requested in an application, a new instance of **HttpApplication** is created. However, to maximize performance, **HttpApplication** instances might be reused for multiple requests. When an instance of **HttpApplication** is created, any configured modules are also created. For instance, if the application is configured to do so, ASP.NET creates a SessionStateModule module. After all configured modules are created, the **HttpApplication** class's Init method is called. The following diagram illustrates this relationship:

	 Client Client Client 1 2 3 Internet IIS (or other Web server) ASP.NET ASP.NET ISAPI Application Domain HttpRuntime HttpContext HttpRequest HttpResponse HttpApplication
The request is processed by the **HttpApplication** pipeline.	The following events are executed by the **HttpApplication** class while the request is processed. The events are of particular interest to developers who want to extend the **HttpApplication** class. 1. Validate the request, which examines the information sent by the browser and determines whether it contains potentially malicious markup. For more information, see ValidateRequest and Script Exploits Overview. 2. Perform URL mapping, if any URLs have been configured in the UrlMappingsSection section of the Web.config file. 3. Raise the BeginRequest event. 4. Raise the AuthenticateRequest event. 5. Raise the PostAuthenticateRequest event. 6. Raise the AuthorizeRequest event. 7. Raise the PostAuthorizeRequest event. 8. Raise the ResolveRequestCache event. 9. Raise the PostResolveRequestCache event. 10. Based on the file name extension of the requested resource (mapped in the application's configuration file), select a class that implements IHttpHandler to process the request. If the request is for an object (page) derived from the Page class and the page needs to be compiled, ASP.NET compiles the page before creating an instance of it. 11. Raise the PostMapRequestHandler event. 12. Raise the AcquireRequestState event. 13. Raise the PostAcquireRequestState event. 14. Raise the PreRequestHandlerExecute event.

	15. Call the ProcessRequest method (or the asynchronous version BeginProcessRequest) of the appropriate **IHttpHandler** class for the request. For example, if the request is for a page, the current page instance handles the request. 16. Raise the PostRequestHandlerExecute event. 17. Raise the ReleaseRequestState event. 18. Raise the PostReleaseRequestState event. 19. Perform response filtering if the Filter property is defined. 20. Raise the UpdateRequestCache event. 21. Raise the PostUpdateRequestCache event. 22. Raise the EndRequest event.

Life Cycle Events and the Global.asax file

During the application life cycle, the application raises events that you can handle and calls particular methods that you can override. To handle application events or methods, you can create a file named Global.asax in the root directory of your application.

If you create a Global.asax file, ASP.NET compiles it into a class derived from the **HttpApplication** class, and then uses the derived class to represent the application.

An instance of **HttpApplication** processes only one request at a time. This simplifies application event handling because you do not need to lock non-static members in the application class when you access them. This also allows you to store request-specific data in non-static members of the application class. For example, you can define a property in the Global.asax file and assign it a request-specific value.

ASP.NET automatically binds application events to handlers in the Global.asax file using the naming convention **Application**_*event*, such as **Application_BeginRequest**. This is similar to the way that ASP.NET page methods are automatically bound to events, such as the page's **Page_Load** event. For details, see ASP.NET Page Life Cycle Overview.

The **Application_Start** and **Application_End** methods are special methods that do not represent **HttpApplication** events. ASP.NET calls them once for the lifetime of the application domain, not for each **HttpApplication** instance.

The following table lists some of the events and methods that are used during the application life cycle. There are many more events than those listed, but they are not commonly used.

Event or method	**Description**
Application_Start	Called when the first resource (such as a page) in an ASP.NET application is requested. The **Application_Start** method is called only one time during the life cycle of an application. You can use this method to perform startup tasks such as loading data into the cache and initializing static values. You should set only static data during application start. Do not set any instance data because it will be available only to the first instance of the **HttpApplication** class that is created.
Application_event	Raised at the appropriate time in the application life cycle, as listed in the application life cycle table earlier in this topic. **Application_Error** can be raised at any phase in the application life cycle. **Application_EndRequest** is the only event that is guaranteed to be raised in every request, because a request can be short-circuited. For example, if two modules handle the **Application_BeginRequest** event and the first one throws an exception, the **Application_BeginRequest** event will not be called for the second module. However, the **Application_EndRequest** method is always called to allow the application to clean up resources.
HttpApplication.Init	Called once for every instance of the **HttpApplication** class after all modules have been created.
Dispose	Called before the application instance is destroyed. You can use this method to manually release any unmanaged resources. For more information, see Cleaning Up Unmanaged Resources.
Application_End	Called once per lifetime of the application before the application is unloaded.

Compilation Life Cycle

When the first request is made to an application, ASP.NET compiles application items in a specific order. The first items to be compiled are referred to as the top-level items. After the first request, the top-level items are recompiled only if a dependency changes. The following table describes the order in which ASP.NET top-level items are compiled.

Item	**Description**
App_GlobalResources	The application's global resources are compiled and a resource assembly is built. Any assemblies in the application's Bin folder are linked to the resource assembly.
App_WebResources	Proxy types for Web services are created and compiled. The resulting Web references assembly is linked to the resource assembly if it exists.
Profile properties defined in the Web.config file	If profile properties are defined in the application's Web.config file, an assembly is generated that contains a profile object.
App_Code	Source code files are built and one or more assemblies are created. All code assemblies and the profile assembly are linked to the resources and Web references assemblies if any.
Global.asax	The application object is compiled and linked to all of the previously generated assemblies.

Once the application's top level items have been compiled, ASP.NET compiles folders, pages, and other items as needed. The following table describes the order in which ASP.NET folders and items are compiled.

Item	**Description**
App_LocalResources	If the folder containing the requested item contains an App_LocalResources folder, the contents of the local resources folder are compiled and linked to the global resources assembly.
Individual Web pages (.aspx files), user controls (.ascx files), HTTP handlers (.ashx files), and HTTP modules (.asmx files)	Compiled as needed and linked to the local resources assembly and the top-level assemblies.
Themes, master pages, other source files	Skin files for individual themes, master pages, and other source code files referenced by pages are compiled when the referencing page is compiled.

Compiled assemblies are cached on the server and reused on subsequent requests and are preserved across application restarts as long as the source code is unchanged.

Because the application is compiled on the first request, the initial request to an application can take significantly longer than subsequent requests. You can precompile your application to reduce the time required for the first request. For more information, see How to: Precompile ASP.NET Web Sites.

Application Restarts
Modifying the source code of your Web application will cause ASP.NET to recompile source files into assemblies. When you modify the top-level items in your application, all other assemblies in the application that reference the top-level assemblies are recompiled as well.

In addition, modifying, adding, or deleting certain types of files within the application's known folders will cause the application to restart. The following actions will cause an application restart:

- Adding, modifying, or deleting assemblies from the application's Bin folder.
- Adding, modifying, or deleting localization resources from the App_GlobalResources or App_LocalResources folders.
- Adding, modifying, or deleting the application's Global.asax file.
- Adding, modifying, or deleting source code files in the App_Code directory.
- Adding, modifying, or deleting Profile configuration.
- Adding, modifying, or deleting Web service references in the App_WebReferences directory.
- Adding, modifying, or deleting the application's Web.config file.

When an application restart is required, ASP.NET will serve all pending requests from the existing application domain and the old assemblies before restarting the application domain and loading the new assemblies.

HTTP Modules

The ASP.NET application life cycle is extensible through IHttpModule classes. ASP.NET includes several classes that implement **IHttpModule**, such as the **SessionStateModule** class. You can also create your own classes that implement **IHttpModule**.

If you add modules to your application, the modules themselves can raise events. The application can subscribe to in these events in the Global.asax file by using the convention *modulename_eventname*. For example, to handle the Authenticate event raised by a FormsAuthenticationModule [http://msdn2.microsoft.com/en-us/library/system.web.security.formsauthenticationmodule.aspx] object, you can create a handler named **FormsAuthentication_Authenticate**.

The **SessionStateModule** class is enabled by default in ASP.NET. All session events are automatically wired up as **Session**_*event*, such as **Session_Start**. The Start event is raised each time a new session is created. For more information, see Session State Overview .

ASP.NET

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. Additionally, 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 any control behavior code. (The life cycle of a control is based on the page life cycle, but the page raises more events for a control than are available for an ASP.NET page alone.)

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 ASP.NET Application Life Cycle Overview.

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 step, 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. Additionally, during the start step, the page's UICulture property is set.
Page initialization	During page initialization, controls on the page are available and each control's UniqueID property is set. Any themes are also applied to the page. 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.
Validation	During validation, the Validate method of all validator controls is called, which sets the IsValid property of individual validator controls and of the page.
Postback event handling	If the request is a postback, any event handlers are called.
Rendering	Before rendering, view state is saved for the page and all controls. During the rendering phase, the page calls the Render method for each control, providing a text writer that writes its output to the OutputStream of the page's **Response** property.
Unload	Unload is called 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 any 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** (or if it is not defined, since by default it is **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 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 your own ASP.NET server controls, you need to understand more about these stages. For information about creating custom controls, see Developing Custom ASP.NET Server Controls.

Page Event	Typical Use
PreInit	Use this event for the following: Check the **IsPostBack** property to determine whether this is the first time the page is being processed. Create or re-create dynamic controls. Set a master page dynamically. Set the Theme property dynamically. Read or set profile property values. **Note** 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.
Init	Raised after all controls have been initialized and any skin settings have been applied. Use this event to read or initialize control properties.
InitComplete	Raised by the Page object. Use this event for processing tasks that require all initialization be complete.
PreLoad	Use this event if you need to perform processing on your page or control before the Load event. After the **Page** raises this event, it loads view state for itself and all controls, and then processes any postback data included with the **Request** instance.
Load	The **Page** calls the OnLoad event method on the **Page**, then recursively does the same for each child control, which does the same for each of its child controls until the page and all controls are loaded. Use the **OnLoad** event method to set properties in controls and establish database connections.
Control events	Use these events to handle specific control events, such as a Button control's Click event or a TextBox control's TextChanged event. **Note** 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.
LoadComplete	Use this event for tasks that require that all other controls on the page be loaded.
PreRender	Before this event occurs: The **Page** object calls EnsureChildControls for each control and for the page. Each data bound control whose DataSourceID property is set calls its DataBind method. For more information, see Data Binding Events for Data-Bound Controls below.

	The **PreRender** event occurs for each control on the page. Use the event to make final changes to the contents of the page or its controls.
SaveStateComplete	Before this event occurs, ViewState has been saved for the page and for all controls. Any changes to the page or controls at this point will be ignored. Use this event perform tasks that require view state to be saved, but that do not make any changes to controls.
Render	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 that is sent to the browser. 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. A user control (an .ascx file) automatically incorporates rendering, so you do not need to explicitly render the control in code.
Unload	This event occurs for each control and then for the page. In controls, use this event to do final cleanup for specific controls, such as closing control-specific database connections. 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. **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).

You can customize the appearance or content of a control by handling the events for the control, such as the **Click** event for the **Button** control and the SelectedIndexChanged event for the ListBox control. Under some circumstances, you might also handle a control's DataBinding or DataBound events. For more information, see the class reference topics for individual controls and Developing Custom ASP.NET Server Controls.

When inheriting a class 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 creating an event handler 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.

Catch-up Events for Added Controls

If controls are created dynamically at run time or are authored 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.

In general, you do not need to be concerned about this unless you have 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 that displays a company record in each row along with 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 CompanyID 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 .

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	This event is raised by data-bound controls before the **PreRender** event of the containing control (or of the **Page** object) and marks the beginning of binding the control to the data. Use this event to manually open database connections, if required. (The data source controls often make this unnecessary.)
RowCreated (**GridView** only) or ItemCreated (DataList , **DetailsView**, SiteMapPath , DataGrid , **FormView**, and controls)	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.
RowDataBound (**GridView** only) or ItemDataBound (**DataList**, **SiteMapPath**, **DataGrid**, and **Repeater** controls)	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 for displaying related data within the row or item.
DataBound	This event marks 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. 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 details, see "Catch-up Events for Added Controls" earlier in this topic.)

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	This event is raised during a postback, after the page's **LoadComplete** event has occurred. It marks the beginning of the login process. Use this event for tasks that must occur prior to beginning the authentication process.
Authenticate	This event is raised after the **LoggingIn** event. Use this event to override or enhance the default authentication behavior of a **Login** control.
LoggedIn	This event is raised after the user name and password have been authenticated. 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.
LoginError	This event is raised if authentication was not successful. Use this event to set text in the control that explains the problem or to direct the user to a different page.