# 1

# The ASP.NET 2.0 Framework



This chapter begins with the following definition of the ASP.NET Framework:

> *ASP.NET is a Framework that processes requests for Web resources.*

In other words, ASP.NET is a request processing architecture or framework. This description of the Framework prompts you to think, "If ASP.NET is a request processing architecture or framework, every component of the framework must exist for one reason and one reason only; that is, to contribute one way or another to the process of handling requests for Web resources."

Think of the components of the ASP.NET Framework in terms of their roles in the overall request handling process. Instead of asking, "What does this component do?" you should ask, "What does this component do to *help process the request?*"

Therefore, this chapter follows a request for a resource from the time it arrives in the Web server (IIS) all the way through the ASP.NET request processing architecture to identify components of the framework that participate or contribute directly or indirectly to the request handling process. However, to keep the discussions simple and focused, the details of these components are left to the following chapters. In short, the main goal of this chapter is to help you understand the big picture.

## Following the Request

To make the discussion more concrete, consider the request for a simple Web page named `Default.aspx`, shown in Listing 1-1. This page consists of a textbox and a button. The user enters a name and clicks the button to post the page back to the server where the name is processed.

**Listing 1-1: The Default.aspx page**

```
<%@ Page Language="C#" AutoEventWireup="true" CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
  <form id="form1" runat="server">
    <strong>Name: </strong>
    <asp:TextBox runat="server" ID="NameTextBox" />
    <br /> <br />
    <asp:Button runat="server" OnClick="SubmitCallback" Text="Submit" />
  </form>
</body>
</html>
```

Figure 1-1 follows the request all the way through the request processing architecture. The block arrows represent the request as it passes through different components. Everything starts when the user attempts to download the `Default.aspx` page; that is, the user makes a request for this page.

As Figure 1-1 shows, Internet Information Services (IIS) picks up the request and passes it to an ISAPI extension named `aspnet_isapi.dll`. After passing through some intermediary components, the request finally arrives in a .NET component named `HttpRuntime`. This component creates an instance of a .NET class named `HttpContext` that contains the complete information about the request, and exposes this information in the form of well-known, convenient managed objects such as `Request`, `Response`, `Server`, and so on. The cylinders in Figure 1-1 represent the `HttpContext` object as it's passed from one component to another.

The request finally arrives in a .NET component named `HttpApplication`. This component creates a pipeline of .NET components known as HTTP modules to perform preprocessing operations such as authentication and authorization. The results of these operations are stored in the `HttpContext` object.

After HTTP modules preprocess the request, `HttpApplication` passes the preprocessed request to a component known as an HTTP handler. The main responsibility of an HTTP handler is to generate the markup text that is sent back to the requester as part of the response. Different types of HTTP handlers generate different types of markup texts. For example, the HTTP handler that handles the requests for the `.aspx` extension, such as the `Default.aspx` page shown in Listing 1-1, generates HTML markup text similar to Listing 1-2. The HTTP handler that handles the requests for the `.asmx` extension, on the other hand, generates XML markup text.

**Listing 1-2: The markup text that the HTTP handler generates in response to a request for the Default.aspx page**

```
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
  <form name="form1" method="post" action="Default.aspx" id="form2">
    <strong>Name: </strong>
    <input name="NameTextBox" type="text" id="Text1" />
    <br /><br />
    <input type="submit" name="Button1" value="Submit" id="Submit1" />
  </form>
</body>
</html>
```

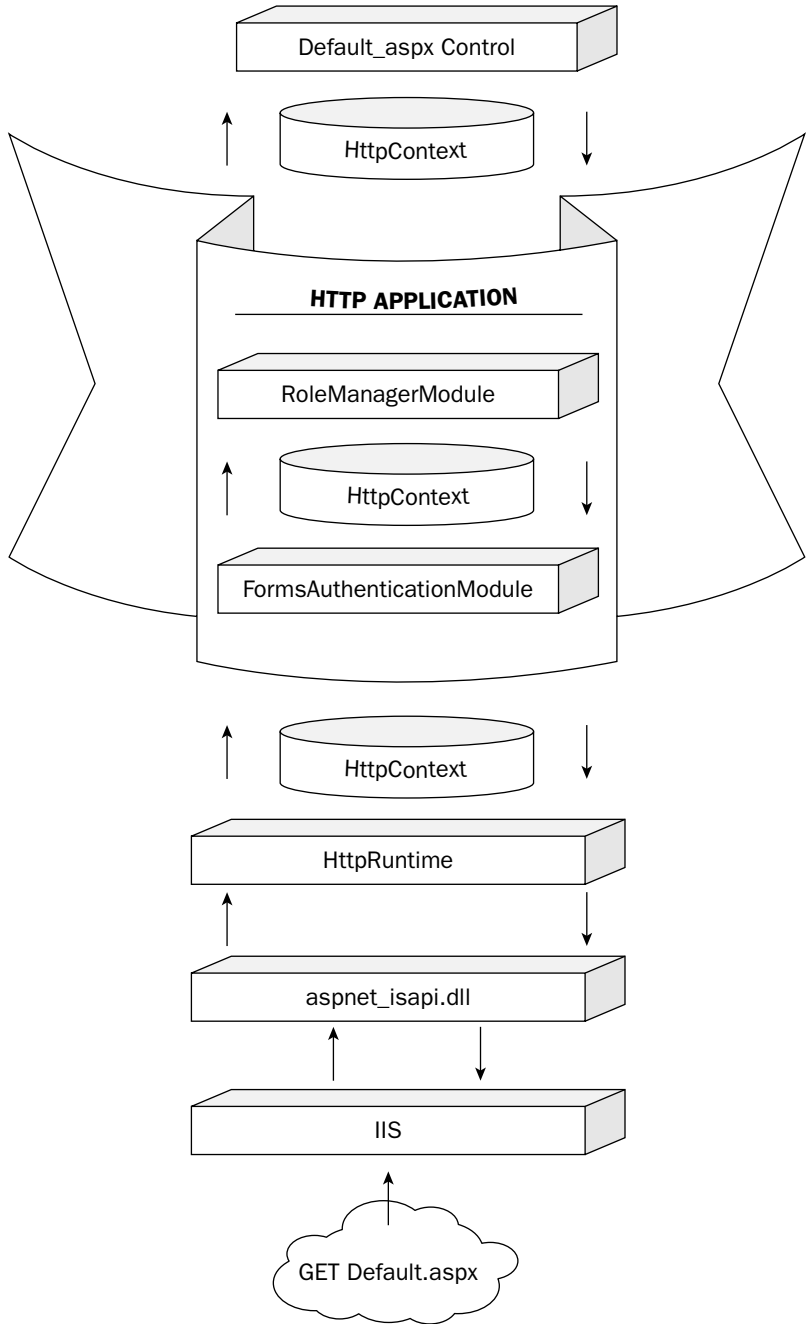
Default_aspx Control
HttpContext
HTTP APPLICATION
RoleManagerModule
HttpContext
FormsAuthenticationModule
HttpContext
HttpRuntime
aspnet_isapi.dll
IIS
GET Default.aspx

Figure 1-1

Therefore, ASP.NET instantiates and initializes the appropriate HTTP handler to handle or process the request for the specified extension. For example, as Figure 1-2 shows, ASP.NET takes the following actions to instantiate and initialize the HTTP handler that handles the request for the `Default.aspx` page; that is, the handler that emits or generates the HTML markup text shown in Listing 1-2 out of the `Default.aspx` file:

1. Parses the `Default.aspx` file.
2. Uses the parsed information to dynamically implement a control or class that derives from the `Page` class. By default, ASP.NET concatenates the name of the `.aspx` file with the string `aspx` and uses it as the name of this dynamically generated class. For example, in the case of the `Default.aspx` file, this class is named `Default_aspx`.

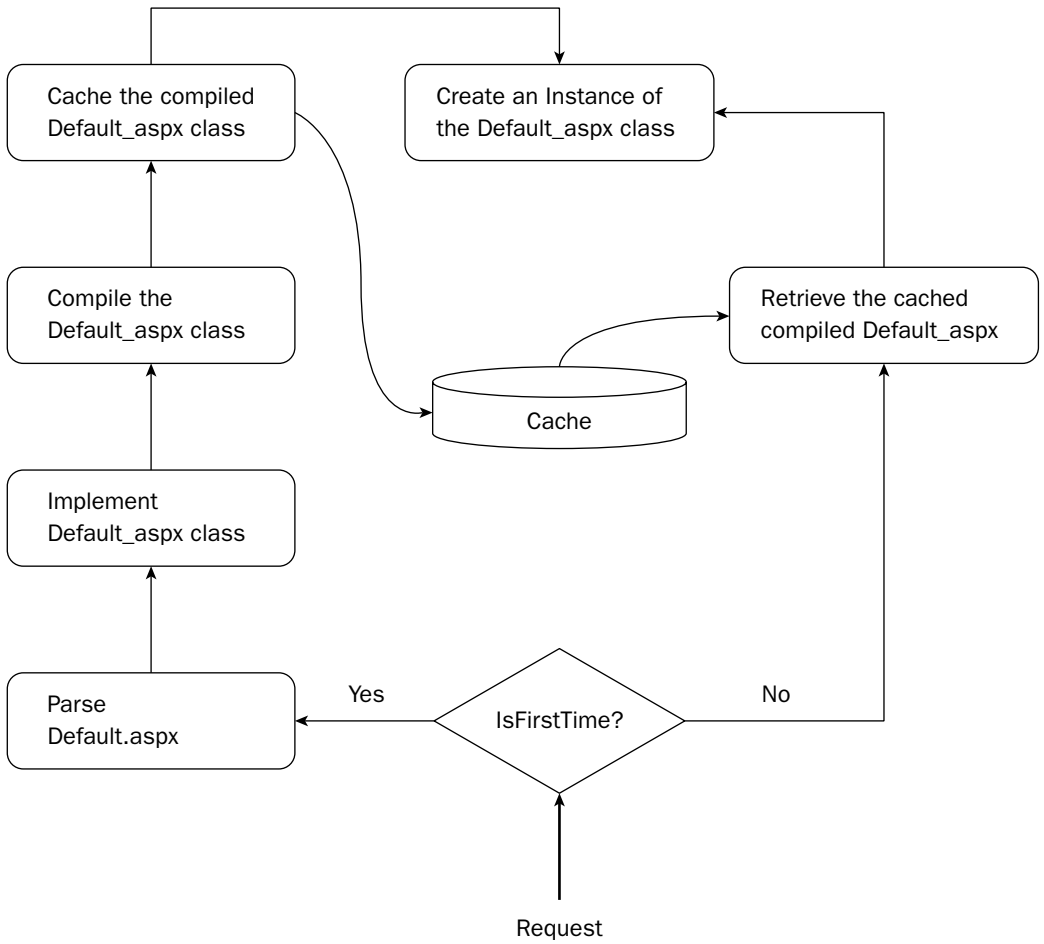


Figure 1-2

*If you're curious as to what the code for this class looks like, here is what you need to do to view the code. Introduce a syntax error in the* `Default.aspx.cs` *and set the* `Debug` *attribute on the* `Page` *directive to true to run the application in debug mode. Now if you access the* `Default.aspx` *page, you'll get a page with a link named "Show Complete Compilation Source." Click this link to view the code for the dynamically generated class. Notice that the class is named* `Default_aspx`*.*

3. Dynamically compiles the control or class.
4. Caches the compiled class.
5. Dynamically creates an instance of the compiled control.
6. Assigns the task of handling the request for the `Default.aspx` file to this instance.

The HTTP handler that processes the request for the `Default.aspx` file is a control or class named `Default_aspx` that ASP.NET dynamically implements from the contents of the `Default.aspx` file. As Figure 1-2 shows, the first four actions are taken only the very first time `Default.aspx` is being accessed. To handle the subsequent requests, ASP.NET simply retrieves the cached compiled `Default_aspx` class from the ASP.NET `Cache` object, creates an instance of the class, and assigns the task of handling the request to this instance.

# Why Develop Custom Components?

As the previous section shows the request for a resource goes through different components of the ASP.NET request processing architecture, where each component contributes toward the final markup text that is sent back to the requester as part of the response message. As discussed, the content or type of the final markup text depends on the type of the requested resource, which normally depends on its extension. For example, the final markup text contains HTML if the request was made for an `.aspx` extension, and XML if the request was made for an `.asmx` extension.

The ASP.NET request processing architecture contains two groups of components:

- Components that directly contribute to the generation of the final markup text. This group includes server controls such as `TextBox`, `LiteralControl`, `Xml`, and so on.
- Components that indirectly contribute to the generation of the final markup text.

As an example of a component that contributes indirectly, consider the `RoleManagerModule` HTTP module shown in Figure 1-1. As you'll see later in this book, at the end of each request, this module retrieves the roles the current user is in from an object known as `principal`, and stores them in a cookie that is sent back to the requesting browser as part of the response.

At the beginning each request, on the other hand, the module retrieves the roles from the cookie and caches them in the `principal` object that represents the current user. Storing roles in a cookie improves the overall performance of a Web application because it's much more efficient to retrieve roles from a cookie than a database.

These roles affect the final markup text that is sent back to the user. For example, the application may only allow users in certain roles to access the `Default.aspx` page. Users who are not in one of those roles will receive a different HTML markup text that contains an error message.

You might be wondering why you would want to develop custom components. Why customize when you can use the existing standard ASP.NET controls and components?

This chapter invites you to change the way you think about the ASP.NET controls and components. This new perspective allows you to think of ASP.NET components in terms of their direct or indirect contributions to the final markup text that is sent back to the requester as part of the response message.

In light of this new perspective, you can easily address the preceding question. You need to develop custom components because there are many situations in which the contributions of the existing standard ASP.NET components do not add up to the markup text that would adequately service the request. The following sections provide a few examples of such situations to give you a flavor of what you can achieve with the skills, knowledge, and experience that this book provides you. Keep in mind that the examples and situations here make up only a very small portion of the situations and examples covered in this book.

## Data Source Controls

Developing data-driven Web applications is a huge challenge because data comes from many different types of data stores, such as relational databases, XML documents, flat files, and Web services, to name a few. It's a challenge because different types of data stores use different types of data access APIs.

As you'll see in Chapters 12, 13, 14, and 15, a data source control is a component that acts as an adapter between the data store and your application. Your application can talk to any type of data store as long as there is an adapter for that type of data store. You can't use the same adapter to talk to two different types of data stores, however — each type of data store has its own adapter.

The ASP.NET Framework comes with standard data source controls for a handful of data stores, such as SQL Server and XML documents. This leaves out numerous data stores that your application cannot talk to. This is where you as the developer come into play. If you know how to develop a data source control, you can easily develop a data source control for your favorite data store to allow your application to talk to the data store without changing a single line of code in your application! Chapters 12–15 will help you gain the skills, knowledge, and experience that you need to develop data source controls.

## Role Manager Modules and Principals

As discussed previously, the standard ASP.NET `RoleManagerModule` stores the current user's role information in a cookie to improve the performance of your application. This leaves out those clients that don't support cookies or where cookies are disabled.

Chapters 20 and 24 help you learn to develop a role manager module that can store the current user's role information in other sources, such as the `Session` object. Those chapters also teach you how to implement the `IPrincipal` interface to develop your own custom principals.

## Role Providers

If you're using the ASP.NET 2.0 Framework to develop a brand new Web application, and if your clients don't have any requirements as to where and in what format or schema you should store your application's role information, you can store the role information in the standard SQL Server database named `aspnetdb`. However, if you have to move your existing applications to the new Framework, you can't afford to dump the database where your application has been storing role information for the last few years and start using the `aspnetdb` database. Or if you're using the ASP.NET 2.0 Framework to develop a brand new Web application but you're required to store your application's role information in a non-SQL Server data store such as XML documents or a SQL Server database with a different schema than the `aspnetdb` database, you can't use the `aspnetdb` database.

Chapters 20, 23, and 24 teach you everything you need to develop a role provider that will allow you to integrate your desired role data store (both relational such as SQL Server and Oracle, and non-relational such as XML documents) with the role management features of the ASP.NET 2.0 Framework.

## Membership Providers

Much like the role provider, if you have applications that you need to move from ASP.NET 1.x to 2.0, you can't afford to dump the databases where you have been storing you user membership information for the last few years and jump to the new `aspnetdb` database. Or if you're developing a brand new ASP.NET 2.0 Web application but you're required to store your application's user membership information in a non-SQL Server data store such as XML documents or a SQL Server database with a different schema than the `aspnetdb` database, you can't use the `aspnetdb` database. Chapters 20, 21, and 22 teach you everything you need to develop a membership provider that will allow you to integrate your desired membership data store (both relational and non-relational) with the new ASP.NET 2.0 user membership feature.

## Customizing XML Web Services and Their Clients

The ASP.NET 2.0 XML Web service infrastructure comes with two new exciting features that work together to provide you with amazing customization power. First, it allows you to implement the `IXmlSerializable` interface to take complete control over the serialization and deserialization of your custom components. Second, it allows you to implement your own custom schema importer extension to customize the code for the proxy class. These two features work hand-in-hand to empower you to resolve problems that you couldn't handle otherwise. One of these common problems is the significant degradation of the performance and responsiveness of XML Web services that expose huge amounts of data. Chapter 11 builds on Chapter 10 to teach you how to implement your own custom schema importer extensions to customize the proxy code generation and how to implement the `IXmlSerializable` interface to customize the serialization and deserialization of your custom components.

## Developing Ajax-Enabled Controls and Components

**A**synchronous **J**avaScript **A**nd **X**ml (Ajax) is a popular Web application development approach that allows a Web application to break free from the traditional click-and-wait user interaction pattern characterized by irritating waiting periods. An Ajax-enabled control or component exhibits the following four important characteristics:

- It has rich client-side functionality that uses client-side resources to handle user interactions. Chapter 26 provides you with a detailed step-by-step recipe for using XHTML/HTML, CSS, DOM, and JavaScript client-side technologies to implement the client-side functionality of an Ajax-enabled control or component. Chapter 26 then uses the recipe to develop two Ajax-enabled controls.
- It uses an asynchronous client callback to communicate with the server without having to perform a full-page postback. Chapter 27 shows you how to use the ASP.NET 2.0 client-callback mechanism to develop Ajax-enabled controls that make asynchronous client-callback requests to the server. Chapter 27 then uses the ASP.NET 2.0 client-callback mechanism to implement several Ajax-enabled controls and components.
- It exchanges data with the server in XML format. Chapter 27, 28, and 29 show you how to develop Ajax-enabled controls and components that use XML, DOM, and JavaScript client-side technologies to read the XML data that they receive from the server and to generate the HTML that displays the XML data. These three chapters also implement several Ajax-enabled controls and components that exchange data with the server in XML format.
- Its communication with the server is governed by Ajax patterns. Chapters 28 and 29 discuss the Predictive Fetch, Periodic Refresh, Submission Throttling, and Explicit Submission Ajax patterns in detail and use these patterns to implement several Ajax-enabled controls and components.

## Developing Web Parts Controls

One of the most exciting and anticipated new features of the ASP.NET 2.0 Framework is its rich support for Web Parts. Chapters 28, 29, 30, and 31 build on earlier chapters to provide you with a comprehensive and in-depth coverage of the ASP.NET Web Parts Framework, where you'll learn how to develop your own custom `WebPart`, `EditorPart`, `CatalogPart`, `WebPartZone`, `WebPartManager`, `WebPartVerb`, and `WebPartChrome` components. This book also implements a set of base data-bound Web Parts controls that you can use to develop custom data-bound Web Parts controls that support the following features:

- They can access any type of data store. This is very important in data-driven Web applications where data come from many different types of data stores such as Microsoft SQL Server, Oracle, XML documents, flat files, Web services, and so on.
- Page developers can use them declaratively on an ASP.NET page without writing a single line of code.
- You can develop them with minimal coding effort.
- They automate all data operations such as `Select`, `Delete`, `Update`, and `Insert` without a single line of code from page developers.

## Developing Custom Data Control Fields

Displaying and editing database records are two of the most common operations in data-driven Web applications. These applications use the appropriate user interface to display database records and to allow end users to edit these records. The type of the user interface depends on the type of the data being displayed or edited.

For example, it makes more sense to provide end users with a `CheckBox` control to edit a Boolean field than a `TextBox` control where the user must enter error-prone text such as "true" or "false." As you'll see in Chapter 18, the ASP.NET Framework comes with components known as *data control fields*, where each component is designed to present the end user with the specific type of user interface to display and edit a data field.

However, the existing data control fields leave out lot of common situations that you as the developer have to deal with. For example, one of the common scenarios in most Web applications is to allow users to edit foreign key fields. None of the existing data control fields provides support for foreign key editing. Chapter 18 teaches you everything you need to know to develop your own custom data control fields to deal with these common situations.

## Developing Custom HTTP Handlers and Modules

As discussed, every HTTP handler is specifically designed to handle requests for a specific extension. The ASP.NET Framework comes with standard handlers that handle common extensions such as `.aspx` and `.asmx`. However, there are situations in which you would want to write your own custom HTTP handlers to handle custom extensions.

For example, many Web applications nowadays provide RSS feeds. It makes lot of sense to implement a custom HTTP handler that handles requests for `.rss` extensions. Such a dedicated HTTP handler will allow you to bypass the normal page life cycle and feed RSS right from this dedicated HTTP handler. Chapter 8 shows you how to develop your own custom HTTP handlers and modules.

## Developing Custom Provider-Based Services

ASP.NET 2.0 uses the Provider Pattern to implement most of its new features and services such as membership, role management, profiles, and so on. This exciting pattern allows these features and services to support any type of data store. Chapter 27 shows you how to develop your own provider-based services that can use any type of data store, and you develop a provider-based RSS service that can emit RSS feeds from any type of data store such as Microsoft SQL Server, Oracle, XML documents, file system, and so on.

# Summary

The main goals of this chapter were twofold: to provide you with a brief overview of different components of the ASP.NET Framework, and to make a case as to why you need to extend these components to implement your own custom server controls and components.

Although ASP.NET 2.0 does provide a variety of useful server controls and components, there are numerous situations in which it can't provide enough functionality to fit your needs. This book provides you with the experience, knowledge, and skills you need to develop custom controls and components that tackle these situations. This chapter outlined just a few of the situations where custom controls and components are your best option, and you'll see all of them by the end of the book. The next chapter starts with the most basic custom controls.