



Forms Authentication

In the previous chapter, you learned about the basic structure of ASP.NET security. In this chapter, you will learn how you can authenticate your users using forms authentication. You should use this type of authentication whenever you do not want to use Windows-based accounts in your applications.

In such cases, you need your own authentication infrastructure with a custom login page that validates a user name and password against a custom store such as your own database. This infrastructure then establishes the security context on each request again (in many cases such systems work based on cookies). If you've ever authenticated users with ASP 3.0, you've probably created such authentication mechanisms on your own.

Fortunately, ASP.NET includes a complete infrastructure for implementing such systems. ASP.NET handles the cookies and establishes the security context on each request for you. This infrastructure is called *forms authentication*, and you'll learn how it works in this chapter.

Note The basic forms authentication infrastructure works the same way as in previous versions of ASP.NET. It includes only a handful of new settings in its configuration schema, covered in the section “Configuring Forms Authentication.” If you have experience with ASP.NET 1.x and forms authentication, you can skip this chapter and proceed with Chapter 21.

Introducing Forms Authentication

Forms authentication is a *ticket-based* (also called *token-based*) system. This means when users log in, they receive a ticket with basic user information. This information is stored in an encrypted cookie that's attached to the response so it's automatically submitted on each subsequent request.

When a user requests an ASP.NET page that is not available for anonymous users, the ASP.NET runtime verifies whether the forms authentication ticket is available. If it's not available, ASP.NET automatically redirects the user to a login page. At that moment, it's your turn. You have to create this login page and validate the credentials within this login page. If the user is successfully validated, you just tell the ASP.NET infrastructure about the success (by calling a method of the FormsAuthentication class), and the runtime automatically sets the authentication cookie (which actually contains the ticket) and redirects the user to the originally requested page. With this request, the runtime detects that the authentication cookie with the ticket is available and grants access to the page. You can see this process in Figure 20-1.

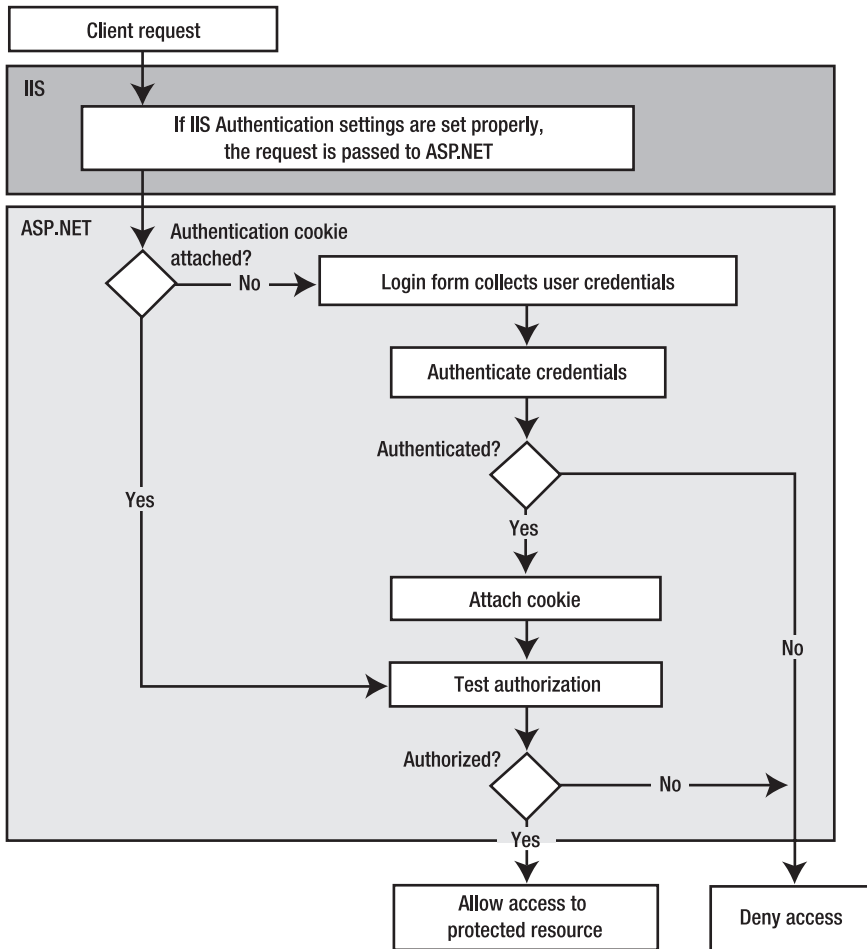


Figure 20-1. *The forms authentication process*

All you need to do is configure forms authentication in the `web.config` file, create the login page, and validate the credentials in the login page.

Why Use Forms Authentication?

Cookie authentication is an attractive option for developers for a number of reasons:

- You have full control over the authentication code.
- You have full control over the appearance of the login form.
- It works with any browser.
- It allows you to decide how to store user information.

Let's look at each of these in turn.

Controlling the Authentication Code

Because forms authentication is implemented entirely within ASP.NET, you have complete control over how authentication is performed. You don't need to rely on any external systems, as you do with Windows or Passport authentication. You can customize the behavior of forms authentication to suit your needs, as you will see in the section “Persistent Cookies in Forms Authentication.”

Controlling the Appearance of the Login Form

You have the same degree of control over the appearance of forms authentication as you do over its functionality. In other words, you can format the login form in any way you like.

This flexibility in appearance is not available in the other authentication methods. Windows authentication needs the browser to collect credentials, and Passport authentication requires that users leave your website and visit the Passport site to enter their credentials.

Working with a Range of Browsers

Forms authentication uses standard HTML as its user interface, so all browsers can handle it. Because you can format the login form in any way you like, you can even use forms authentication with browsers that do not use HTML, such as those on mobile devices. To do this, you need to detect the browser being used and provide a form in the correct format for the device (such as WML for mobile phones).

Caution Forms authentication uses standard HTML forms for collecting and submitting the user's credentials. Therefore, you have to use SSL to encrypt and transmit the user's credentials securely. If you don't use SSL, the information is transmitted as clear text in the postback data in the request to the server.

Storing User Information

Forms authentication stores users in the web.config file by default, but, as you will see in the section “Custom Credentials Store,” you can store the information anywhere you like. You just need to create the code that accesses the data store and retrieves the required information. (And if you use the Membership API introduced in Chapter 21, you even don't need to do that.) A common example is to store the user information in a custom database.

This flexibility in the storage of user information also means you can control how user accounts are created and administered, and you can attach additional information to user accounts.

By comparison, Windows authentication (discussed in Chapter 22) is much less flexible. It requires that you set up a Windows user account for each user you want to authenticate. This is obviously a problem if you want to serve a large number of users or if you want to register users programmatically. It also doesn't allow you to store additional information about users. (In the case of Active Directory, you have the possibility of extending the schema, but this is something that needs to be planned well.) Instead, you have to store this information separately. Passport authentication has similar limitations. Although Passport stores more user information, it doesn't allow you to add custom information, and it doesn't allow you to take part in user registration or account management.

Why Would You *Not* Use Forms Authentication?

So far, you've considered the reasons that make forms authentication an attractive choice for user authentication. However, forms authentication also has downsides:

- You have to create the user interface for users to log in.
- You have to maintain a catalog with user credentials.
- You have to take additional precautions against the interception of network traffic.

The following sections explore these issues. You can solve the first two of these downsides by using the Membership API framework, which offers prebuilt controls and a prebuilt schema for credential storage and runs on SQL Server databases out of the box. You will learn about the Membership API framework in Chapter 21.

Creating Your Own Login Interface

As mentioned earlier, forms authentication gives you control over the interface that users use to log into your web application. Along with its benefits, this approach also creates extra work, because you have to build the login page. Other forms of authentication supply some prebuilt portions. For instance, if you're using Windows authentication, the browser provides a standard dialog box. In Passport authentication, the user interface of the Passport site is always used for logging in.

Creating the login page for forms authentication doesn't require a lot of work, though. It's just worth noting that forms authentication is merely a framework for building an authentication system, rather than an all-in-one system that's complete and ready to use.

The new Membership API, on the other hand, includes a prebuilt login control that can be used either on a separate login page or within any page of your application that provides a prebuilt login user interface. This user interface is customizable and communicates with the Membership API to log the user in automatically. The control does most of the work of creating custom login pages. In most cases, creating a custom login page requires nothing more than adding an .aspx page to your solution with a login control on it. You don't need to catch any events or write any code if you are fine with the default behavior of the control (which will usually be the case). You will learn more details about this control in Chapter 21.

Maintaining User Details

When you use forms authentication, you are responsible for maintaining the details of the users who access your system. The most important details are the credentials that the user needs in order to log into the system. Not only do you need to devise a way to store them, but you also need to ensure that they are stored securely. Also, you need to provide some sort of administration tools for managing the users stored in your custom store.

The Membership API framework ships with a prebuilt schema for storing credentials in a SQL Server database. So, you can save lots of time using this existing schema; furthermore, the schema is extensible. Still, you are responsible for backing up the credentials store securely so that you can restore it in case of a system failure.

All these considerations don't apply to most other types of authentication. In Windows authentication, user credentials are stored by the underlying operating system. Windows uses a variety of techniques to keep them secure automatically so that you don't need to perform any work of your own. In Passport authentication, the credentials are stored securely on Passport servers.

Intercepting Network Traffic

When a user enters credentials for forms authentication, the credentials are sent from the browser to the server in plain-text format. This means anyone intercepting them will be able to read them. This is obviously an insecure situation.

The usual solution to this problem is to use SSL (as described in the previous chapter). Now, a valid argument might be that you just need to use SSL for securing the login page, not the entire application. You can configure forms authentication to encrypt and sign the cookie, and therefore it's extremely difficult for an attacker to get any information from it. In addition, the cookie should not contain any sensitive information and therefore won't include the password that was used for authentication.

But what if the attacker intercepts the unencrypted traffic and just picks the (already encrypted) cookie and uses it for replay? The attacker doesn't need to decrypt it; she just needs to send the cookie with her own request across the wire. You can mitigate such a *replay attack* only if you run the entire website with SSL.

Other authentication mechanisms don't require this extra work. With Windows authentication, you can use a protocol that automatically enforces a secure login process (with the caveat that this is not supported by all browsers and all network environments). With Passport authentication, the login process is handled transparently by the Passport servers, which always use SSL.

Why Not Implement Cookie Authentication Yourself?

Cookie authentication is, on the surface, a fairly straightforward system. You might wonder why you shouldn't just implement it yourself using cookies or session variables.

The answer is the same reason developers don't implement features in ASP.NET ranging from session state to the web control framework. Not only does ASP.NET save you the trouble, but it also provides an implementation that's secure, well tested, and extensible. Some of the advantages provided by ASP.NET's implementation of forms authentication include the following:

- The authentication cookie is secure.
- Forms authentication is a well-tested system.
- Forms authentication integrates with the .NET security classes.

Keeping the Authentication Cookie Secure

Cookie authentication seems simple, but if it's not implemented correctly, you can be left with an insecure system. On their own, cookies are not a safe place to store sensitive information, because a malicious user can easily view and edit cookie data. If your authentication is based on unprotected cookies, attackers can easily compromise your system.

By default, the forms authentication module encrypts its authentication information before placing it in a cookie. It also attaches a hash code and validates the cookies when they return to the server to verify that no changes have been made. The combination of these two processes makes these cookies very secure and saves you from needing to write your own security code. Most examples of homemade cookie authentication are far less secure.

Forms Authentication Is Well Tested

Forms authentication is an integral part of ASP.NET, so it has already been used in a number of web applications and websites. Because so many people use the same system, flaws are quickly discovered, publicized, and solved. As long as you keep up-to-date with patches, you have a high level of

protection. On the other hand, if you create your own cookie authentication system, you do not have the advantage of this widespread testing. The first time you'll notice a vulnerability will probably be when your system is compromised.

Integrating with the ASP.NET Security Framework

All types of ASP.NET authentication use a consistent framework. Forms authentication is fully integrated with this security framework. For example, it populates the security context (IPrincipal) object and user identity (IIdentity) object, as it should. This makes it easy to customize the behavior of forms authentication.

The Forms Authentication Classes

The most important part of the forms authentication framework is the FormsAuthenticationModule, which is an HttpModule class that detects existing forms authentication tickets in the request. If the ticket is not available and the user requests a protected resource, it automatically redirects the request to the login page configured in your web.config file before this protected resource is even touched by the runtime.

If the ticket is present, the module automatically creates the security context by initializing the HttpContext.Current.User property with a default instance of GenericPrincipal, which contains a FormsIdentity instance with the name of the currently logged-in user. Basically, you don't work with the module directly. Your interface to the module consists of the classes in Table 20-1, which are part of the System.Web.Security namespace.

Table 20-1. The Forms Authentication Framework Classes

Class Name	Description
FormsAuthentication	This is the primary class for interacting with the forms authentication infrastructure. It provides basic information about the configuration and allows you to create the ticket, set the cookie, and redirect from the login page to the originally requested page if the validation of credentials was successful.
FormsAuthenticationEventArgs	This module raises an Authenticate event that you can catch. The event arguments passed are encapsulated in an instance of this class. It contains basic information about the authenticated user.
FormsAuthenticationTicket	This class represents the user information that will be encrypted and stored in the authentication cookie.
FormsIdentity	This class is an implementation of IIdentity that is specific to forms authentication. The key addition to the FormsIdentity class is the Ticket property, which exposes the authentication ticket. This allows you to store and retrieve additional information in the ticket.
FormsAuthenticationModule	This is the core of the forms authentication infrastructure that establishes the security context and performs the automatic page redirects to the login page if necessary.

Mostly you will use the FormsAuthentication class and the FormsIdentity class, which represents a successfully authenticated user in your application. Next you will learn how to use forms authentication in your application.

Implementing Forms Authentication

Basically, you need to complete the following steps to use forms authentication in your application:

1. Configure forms authentication in the web.config file.
2. Configure IIS to *allow* anonymous access to the virtual directory, and configure ASP.NET to *restrict* anonymous access to the web application.
3. Create a custom login page that collects and validates a user name and password and then interacts with the forms authentication infrastructure for creating the ticket.

The following sections describe these steps.

Note The cookie is encrypted with a machine-specific key that's defined in the machine.config file. Usually, this detail isn't important. However, in a web farm you need to make sure all servers use the same key so that one server can decrypt the cookie created by another.

Configuring Forms Authentication

You have to configure forms authentication appropriately in your web.config file. Remember from the previous chapter that every web.config file includes the <authentication /> configuration section. Forms authentication works if you configure this section with the value Forms for the mode attribute:

```
<authentication mode="Forms">
  <!-- Detailed configuration options -->
</authentication>
```

The <authentication /> configuration is limited to the top-level web.config file of your application. If the mode attribute is set to Forms, ASP.NET loads and activates the FormsAuthenticationModule, which does most of the work for you. The previous configuration basically uses default settings for forms authentication that are hard-coded into the ASP.NET runtime. You can override any default settings by adding settings to the <system.web> section of the machine.config file. You can override these default settings in your application by specifying additional settings in the <forms /> child tag of this section. The following code snippet shows the complete set of options for the forms tag:

```
<authentication mode="Forms">
  <!-- Detailed configuration options -->
  <forms name="MyCookieName"
    loginUrl="MyLogin.aspx"
    timeout="20"
    slidingExpiration="true"
    cookieless="AutoDetect"
    protection="All"
    requireSSL="false"
    enableCrossAppRedirects="false"
    defaultUrl="MyDefault.aspx"
    domain="www.mydomain.com"
    path="/" />
</authentication>
```

The properties are listed in the order you will use them in most cases. Table 20-2 describes the details of these properties and their default configuration.

Table 20-2. *The Forms Authentication Options*

Option	Default	Description
name	.ASPXAUTH	The name of the HTTP cookie to use for authentication (defaults to .ASPXAUTH). If multiple applications are running on the same web server, you should give each application's security cookie a unique name.
loginUrl	login.aspx	Defines which page the user should be redirected to in order to log into the application. This could be a page in the root folder of the application, or it could be in a subdirectory.
timeout	30	The number of minutes before the cookie expires. ASP.NET will refresh the cookie when it receives a request, as long as half of the cookie's lifetime has expired. The expiry of cookies is a significant concern. If cookies expire too often, users will have to log in often, and the usability of your application may suffer. If they expire too seldom, you run a greater risk of cookies being stolen and misused.
slidingExpiration	false	This attribute enables or disables sliding expiration of the authentication cookie. If enabled, the expiration of an authentication cookie will be reset by the runtime with every request a user submits to the page. This means with every request the expiration of the cookie will be extended.
cookieless	UseDeviceProfile	Allows you to specify whether the runtime uses cookies for sending the forms authentication ticket to the client. Possible options are AutoDetect, UseCookies, UseUri, and UseDeviceProfile.
protection	All	Allows you to specify the level of protection for the authentication cookie. The option All encrypts and signs the authentication cookie. Other possible options are None, Encryption (encrypts only), and Validation (signs only).
requireSSL	false	If set to true, this property has the effect that the browser simply doesn't transmit the cookie if SSL is not enabled on the web server. Therefore, forms authentication will not work in this case if SSL is not activated on the web server.
enableCrossAppRedirects	false	Enables cross-application redirects when using forms authentication for different applications on your server. Of course, this makes sense only if both applications rely on the same credential store and use the same set of users and roles.

Option	Default	Description
defaultUrl	Default.aspx	If the FormsAuthenticationModule redirects a request from the user to the login page, it includes the originally requested page when calling the login page. Therefore, when returning from the login page, the module can use this URL for a redirect after the credentials have been validated successfully. But what if the user browses to the login page directly? This option specifies the page to redirect to if the user accesses the login page directly by typing its URL into the address bar of the browser.
domain	Your host	Specifies the domain for which this cookie is valid. Overriding this property is useful if you want to enable the cookie to be used for more applications on your web server.
path	/	The path for cookies issued by the application. The default value (/) is recommended, because case mismatches can prevent the cookie from being sent with a request.

As explained in Table 20-2, you can disable cookie validation and encryption. However, it's reasonable to wonder why you would want to remove this protection. The only case in which you might make this choice is if you are not authenticating users for security reasons but simply identifying users for personalization purposes. In these cases, it does not really matter if a user impersonates another user, so you might decide that the overhead of encrypting, decrypting, and validating the authentication cookies will adversely affect performance without offering any benefits. Think carefully before taking this approach, however—you should use this approach only in situations where it really does not matter if the authentication system is subverted.

Credentials Store in web.config

When using forms authentication, you have the choice of where to store credentials for the users. You can store them in a custom file or in a database; basically, you can store them anywhere you want if you provide the code for validating the user name and password entered by the user with the values stored in your credential store.

The easiest place to store credentials is directly in the web.config file through the <credentials /> subelement of the <forms /> configuration tag introduced previously.

```
<authentication mode="Forms">
  <!-- Detailed configuration options -->
  <forms name="MyCookieName"
    loginUrl="MyLogin.aspx"
    timeout="20">
    <credentials passwordFormat="Clear">
      <user name="Admin" password="(Admin1)"/>
      <user name="Mario" password="Szpuszta"/>
      <user name="Matthew" password="MacDonald"/>
    </credentials>
  </forms>
</authentication>
```

Note First, using `web.config` as a credential store is possible for simple solutions with just a few users only. In larger scenarios, you should use the Membership API, which is described in Chapter 21. Second, you can hash password values for credentials stored in the `web.config` file. Hashing is nothing more than applying one-way encryption to the password. This means the password will be encrypted in a way that it can't be decrypted anymore. You will learn how you can hash passwords correctly when creating a custom Membership provider in Chapter 26.

Denying Access to Anonymous Users

As mentioned earlier, you do not need to restrict access to pages in order to use authentication. It is possible to use authentication purely for personalization so that anonymous users view the same pages as authenticated users (but see slightly different, personalized content). However, to demonstrate the redirection functionality of forms authentication, it's useful to create an example that denies access to anonymous users. This will force ASP.NET to redirect anonymous users to the login page.

Chapter 23 describes authorization in detail. For now, you'll use the simple technique of denying access to all unauthenticated users. To do this, you must use the `<authorization>` element of the `web.config` file to add a new authorization rule, as shown here:

```
<configuration>
  <system.web>
    <!-- Other settings omitted. -->
    <authorization>
      <deny users="?" />
    </authorization>
  </system.web>
</configuration>
```

The question mark (?) is a wildcard character that matches all anonymous users. By including this rule in your `web.config` file, you specify that anonymous users are not allowed. Every user must be authenticated, and every user request will require the forms authentication ticket (which is a cookie). If you request a page in the application directory now, ASP.NET will detect that the request isn't authenticated and attempt to redirect the request to the login page (which will probably cause an error, unless you've already created this page).

Tip Unlike the `<authentication>` element, the `<authorization>` element is not limited to the `web.config` file in the root of the web application. Instead, you can use it in any subdirectory, thereby allowing you to set different authorization settings for different groups of pages. You'll learn much more about authorization in Chapter 23.

Creating a Custom Login Page

Next, you have to create a custom login page. This page collects a user name and password from the user and validates it against the credentials stored in the credential store. If credentials are stored in `web.config`, this is extremely easy; it's not much harder having credentials stored in any other store such as an external database.

The login page you have to create must contain the parts shown in Figure 20-2. Furthermore, you must include the code for validating the credentials.

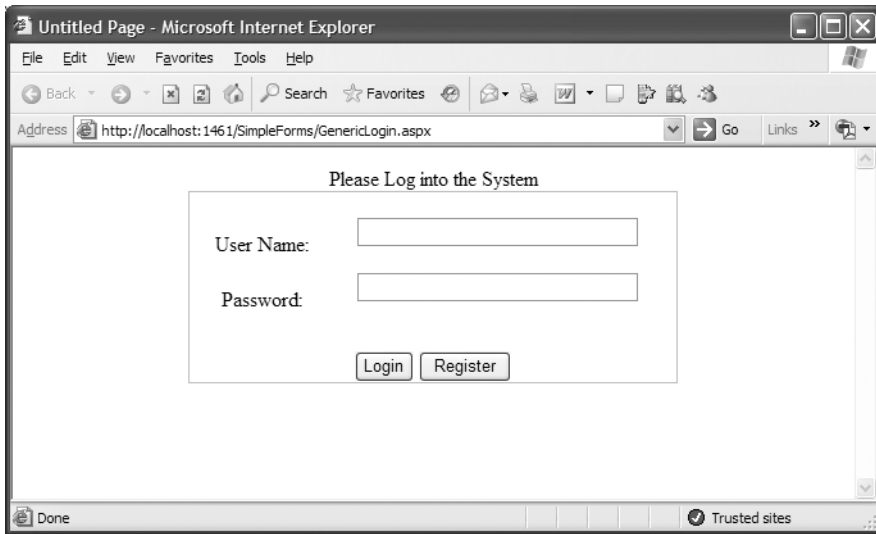


Figure 20-2. A typical login page for a web application

The ASP.NET page shown in Figure 20-2 contains the text boxes for entering the values. Note that the URL includes the originally requested page as a query parameter. This parameter is used by the FormsAuthentication class later for redirecting to the originally requested page. If not present, it uses the page configured in the defaultUrl attribute of the <forms /> configuration tag.

What you cannot see in Figure 20-2 are validation controls. Validation controls are especially important to let the user enter only valid values for a user name and a password. Remember what we mentioned in the previous chapter: never trust user input. Validation adheres to this principle by ensuring that only valid values are entered. Here you can see all the controls contained on the login page:

```
<form id="form1" runat="server">
  <div style="text-align: center">
    Please Log into the System<br />
    <asp:Panel ID="MainPanel" runat="server" Height="90px" Width="380px"
      BorderColor="Silver" BorderStyle="Solid" BorderWidth="1px">
      <br />
      <table width="100%" border="0" cellpadding="0" cellspacing="0">
        <tr>
          <td width="30%" style="height: 43px">
            User Name:</td>
          <td width="70%" style="height: 43px">
            <asp:TextBox ID="UsernameText"
              runat="server" Width="80%" />
            <asp:RequiredFieldValidator
              ID="UsernameRequiredValidator" runat="server"
              ErrorMessage="*" ControlToValidate="UsernameText" />
            <br />
            <asp:RegularExpressionValidator
              ID="UsernameValidator" runat="server"
              ControlToValidate="UsernameText"
              ErrorMessage="Invalid username"
              ValidationExpression="[\\w| ]*" />
          </td>
```

```

</tr>
<tr>
  <td width="30%" style="height: 26px">
    Password:</td>
  <td width="70%" style="height: 26px">
    <asp:TextBox ID="PasswordText" runat="server"
      Width="80%" TextMode="Password" />
    <asp:RequiredFieldValidator ID="PwdRequiredValidator"
      runat="server" ErrorMessage="*"
      ControlToValidate="PasswordText" />
    <br />
    <asp:RegularExpressionValidator ID="PwdValidator"
      runat="server" ControlToValidate="PasswordText"
      ErrorMessage="Invalid password"
      ValidationExpression='[\w| !"$$%&#;/()=\-?\*]*' />
  </td>
</tr>
</table>
<br />
<asp:Button ID="LoginAction" runat="server"
  OnClick="LoginAction_Click" Text="Login" /><br />
<asp:Label ID="LegendStatus" runat="server"
  EnableViewState="false" Text="" />
</asp:Panel>
</div>
</form>

```

As mentioned previously, the validation controls serve two purposes. First, the RequiredFieldValidator controls ensure that both a user name and password are entered in a valid format containing only the characters allowed for user names and passwords. Second, the RegularExpressionValidator controls ensure that only valid values are entered in the User Name text field and in the Password text field. For example, the user name may contain letters, digits, and spaces only. Therefore, the validation expression looks like this:

```
ValidationExpression="[\w| ]*"
```

The \w character class is equivalent to [a-zA-Z_0-9], and the space afterward allows spaces in the user name. The password, for example, may also contain special characters. Therefore, the validation expression looks different from the previous one, as shown here:

```
ValidationExpression='[\w| !"$$%&#;/()=\-?\*]*'
```

Note that the single quote is used for enclosing the attribute value, because this uses the double quote as the allowed special character. Furthermore, because the attribute is contained in the tag code (and therefore the HTML entity), & indicates that the ampersand (&) character is allowed in the password. You can see the validation controls in action in Figure 20-3.

As you can see in Figure 20-3, with validation controls in place you can stop users from entering values for the user name or password that would lead to a SQL injection attack. In addition to using parameterized SQL queries (introduced in Chapter 7), you should always use validation controls to mitigate this type of attack in your applications.

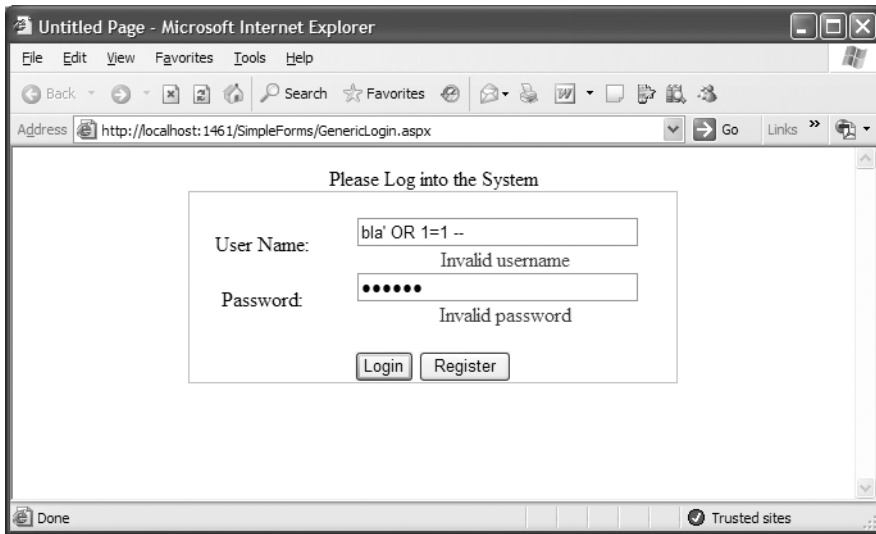


Figure 20-3. Validation controls in action

The last step for creating the login page is to write the code for validating the credentials against the values entered by the user. You have to add the necessary code to the Click event of the login button. Because the following Click event is using the credentials store of the web.config file, validation is fairly easy:

```
protected void LoginAction_Click(object sender, EventArgs e)
{
    Page.Validate();
    if (!Page.IsValid) return;

    if (FormsAuthentication.Authenticate(UsernameText.Text, PasswordText.Text))
    {
        // Create the ticket, add the cookie to the response,
        // and redirect to the originally requested page
        FormsAuthentication.RedirectFromLoginPage(UsernameText.Text, false);
    }
    else
    {
        // User name and password are not correct
        LegendStatus.Text = "Invalid username or password!";
    }
}
```

Note Because forms authentication uses standard HTML forms for entering credentials, the user name and password are sent over the network as plain text. This is an obvious security risk—anyone who intercepts the network traffic will be able to read the user names and passwords that are entered into the login form. For this reason, it is strongly recommended that you encrypt the traffic between the browser and the server using SSL (as described in Chapter 19), at least while the user is accessing the login page.

Furthermore, it's important to include the `Page.IsValid` condition at the beginning of this procedure. The reason for this is that validation controls by default use JavaScript for client-side validation. When calling `Page.Validate()`, the validation takes place on the server. This is important for browsers that either have JavaScript turned off or don't support it. Therefore, if you don't include this part, validation will not happen if the browser doesn't support JavaScript or doesn't have JavaScript enabled. So, you should always include server-side validation in your code.

The `FormsAuthentication` class provides two methods that are used in this example. The `Authenticate()` method checks the specified user name and password against those stored in the `web.config` file and returns a Boolean value indicating whether a match was found. Remember that the methods of `FormsAuthentication` are static, so you do not need to create an instance of `FormsAuthentication` to use them—you simply access them through the name of the class.

```
if (FormsAuthentication.Authenticate(UsernameText.Text, PasswordText.Text))
```

If a match is found for the supplied credentials, you can use the `RedirectFromLoginPage()` method, as shown here:

```
FormsAuthentication.RedirectFromLoginPage(UsernameText.Text, false);
```

This method performs several tasks at once:

1. It creates an authentication ticket for the user.
2. It encrypts the information from the authentication ticket.
3. It creates a cookie to persist the encrypted ticket information.
4. It adds the cookie to the HTTP response, sending it to the client.
5. It redirects the user to the originally requested page (which is contained in the query string parameter of the login page request's URL).

The second parameter of `RedirectFromLoginPage()` indicates whether a persistent cookie should be created. Persistent cookies are stored on the user's hard drive and can be reused for later visits. Persistent cookies are described in the section "Persistent Cookies in Forms Authentication" later in this chapter.

Finally, if `Authenticate()` returns false, an error message is displayed on the page. Feedback such as this is always useful. However, make sure it doesn't compromise your security. For example, it's all too common for developers to create login pages that provide separate error messages depending on whether the user has entered a user name that isn't recognized or a correct user name with the wrong password. This is usually not a good idea. If a malicious user is trying to guess a user name and password, the user's chances increase considerably if your application gives this sort of specific feedback.

Logging Out

Logging a user out of forms authentication is as simple as calling the `FormsAuthentication.SignOut()` method. You can create a logout button and add this code, as shown here:

```
protected void SignOutAction_Click(object sender, EventArgs e)
{
    FormsAuthentication.SignOut();
    FormsAuthentication.RedirectToLoginPage();
}
```

When you call the `SignOut()` method, you remove the authentication cookie. Depending on the application, you may want to redirect the user to another page when the user logs out. If the user requests another restricted page, the request will be redirected to the login page. You can also

redirect to the login page immediately after calling the sign-out method. Or you can use the `Response.Redirect` method.

Tip In a sophisticated application, your login page might not actually be a page at all. Instead, it might be a separate portion of the page—either a distinct HTML frame or a separately coded user control. Using these techniques, you can keep a login and logout control visible on every page. The Membership API framework includes ready-to-use controls for providing this type of functionality.

Hashing Passwords in web.config

Forms authentication includes the possibility of storing the password in different formats. In the `<credentials />` configuration section, the format of the password is specified through the `passwordFormat` attribute, which has three valid values:

- **Clear:** The passwords are stored as clear text in the `<user />` elements of the `<credentials />` section.
- **MD5:** The hashed version of the password is stored in the `<user />` elements, and the algorithm used for hashing the password is the MD5 hashing algorithm.
- **SHA1:** The `<user />` elements in the `<credentials />` section of the `web.config` file contain the hashed password, and the algorithm used for hashing the password is the SHA1 algorithm.

When using the hashed version of the passwords, you have to write a tool or some code that hashes the passwords for you and stores them in the `web.config` file. For storing the password, you should then use the `FormsAuthentication.HashForStoringInConfigFile` method instead of passing in the clear-text password as follows:

```
string hashedPwd = FormsAuthentication.HashForStoringInConfigFile(
    clearTextPassword, "SHA1");
```

The first parameter specifies the clear-text password, and the second one specifies the hash algorithm you should use. The result of the method call is the hashed version of the password.

If you want to modify users stored in `web.config` as shown previously, you have to use the configuration API of the .NET Framework. You cannot edit this section with the web-based configuration tool. The following code snippet shows how you can modify the section through the configuration API:

```
Configuration MyConfig = WebConfigurationManager.OpenWebConfiguration("~/");

ConfigurationSectionGroup SystemWeb = MyConfig.SectionGroups["system.web"];
AuthenticationSection AuthSec =
    (AuthenticationSection)SystemWeb.Sections["authentication"];
AuthSec.Forms.Credentials.Users.Add(
    new FormsAuthenticationUser(UserText.Text, PasswordText.Text));

MyConfig.Save();
```

Of course, only privileged users such as website administrators should be allowed to execute the previous code, and the process executing the code must have write access to your `web.config` file. Also, this sort of code should not be included in the actual web application. You should include it in an administration application only. You will learn more about hashing passwords in Chapters 25 and 26.

Cookieless Forms Authentication

New to ASP.NET 2.0 is that the runtime supports cookieless forms authentication out of the box. In ASP.NET 1.x you had to write this functionality on your own. If you don't want the runtime to use cookies, you configure this through the cookieless attribute of the <forms /> tag in the <authentication /> section.

```
<authentication mode="Forms">
  <!-- Detailed configuration options -->
  <forms name="MyCookieName"
        loginUrl="MyLogin.aspx"
        cookieless="AutoDetect" />
</authentication>
```

The cookieless option includes the possible settings in Table 20-3.

Table 20-3. *Cookieless Options in the <forms /> Configuration*

Option	Description
UseCookies	Forces the runtime to use cookies when working with forms authentication. This requires the client browser to support cookies.
UseUri	If this configuration option is selected, cookies will not be used for authentication. Instead, the runtime encodes the forms authentication ticket into the request URL, and the infrastructure processes this specific portion of the URL for establishing the security context.
AutoDetect	Results in the use of cookies if the client browser supports cookies. Otherwise, URL encoding of the ticket will be used. This is established through a probing mechanism.
UseDeviceProfile	Results in the use of cookies or URL encoding based on a device profile configuration stored on the web server. These profiles are stored in .browser files in the <drive>:\<windows directory>\Microsoft.NET\Framework\v2.0.50215\CONFIG\Browsers directory.

Custom Credentials Store

As mentioned previously, the credential store in web.config is useful for simple scenarios only. You won't want to use web.config as the credential store for a number of reasons:

- **Potential lack of security:** Even though users aren't able to directly request the web.config file, you may still prefer to use a storage medium where you can secure access more effectively. As long as this information is stored on the web server, passwords are accessible to any administrator, developer, or tester who has access.
- **No support for adding user-specific information:** For example, you might want to store information such as addresses, credit cards, personal preferences, and so on.
- **Poor performance with a large number of users:** The web.config file is just a file, and it can't provide the efficient caching and multiuser access of a database.

Therefore, in most applications you will use your own custom credential store for user name and password combinations, and mostly it will be a database such as SQL Server. In ASP.NET 1.x, you had to implement this scenario on your own. In your login form you then had to connect to the database, verify whether the user exists, compare the password stored in the database to the one entered by the user, and then call FormsAuthentication.RedirectFromLoginPage if the user name

and password entered by the user were valid. The following example demonstrates this, and it assumes that you have written a function `MyAuthenticate` that connects to a SQL Server database and reads the corresponding user entry. It returns `true` if the entered user name and password match the ones stored in the database.

```
protected void LoginAction_Click(object sender, EventArgs e)
{
    Page.Validate();
    if (!Page.IsValid) return;

    if (this.MyAuthenticate(UsernameText.Text, PasswordText.Text))
    {
        FormsAuthentication.RedirectFromLoginPage(UsernameText.Text, false);
    }
    else
    {
        LegendStatus.Text = "Invalid username or password!";
    }
}
```

Fortunately, ASP.NET 2.0 provides a ready-to-use infrastructure as well as a complete set of security-related controls that do this for you. The Membership API includes a SQL Server–based data store for storing users and roles and functions for validating user names and passwords against users of this store without knowing any details about the underlying database, as you will learn in Chapter 21. Furthermore, this infrastructure is completely extensible through custom providers, as you will learn in Chapter 26.

Persistent Cookies in Forms Authentication

The examples you've seen so far have used a nonpersistent authentication cookie to maintain the authentication ticket between requests. This means that if the user closes the browser, the cookie is immediately removed. This is a sensible step that ensures security. It's particularly important with shared computers to prevent another user from using a previous user's ticket. Nonpersistent cookies also make *session hijacking* attacks (where a malicious user gains access to the network and steals another user's cookie) more difficult and more limited.

Despite the increased security risks of using persistent authentication cookies, it is appropriate to use them in certain situations. If you are performing authentication for personalization rather than for controlling access to restricted resources, you may decide that the usability advantages of not requiring users to log in on every visit outweigh the increased danger of unauthorized use.

Once you have decided to use persistent cookies, implementing them is easy. You simply need to supply a value of `true` rather than `false` for the second parameter of the `RedirectFromLoginPage()` or `SetAuthCookie()` method of the `FormsAuthentication` class. Here's an example:

```
FormsAuthentication.RedirectFromLoginPage(User nameTextBox.Text, true);
```

By default, persistent cookies do not expire unless the `FormsAuthentication.SignOut()` method is used. Persistent cookies are not affected by the timeout attribute that is set in the `<forms>` element of the web.config file. If you want the persistent cookie to eventually expire sometime in the future, you have to use the `GetAuthCookie()` method of `FormsAuthentication`, set the expiry date and time, and then write the cookie to the HTTP response yourself.

The following example rewrites the code that authenticates the user when the login button is clicked. It creates a persistent cookie but performs additional steps to limit the cookie's life span to ten days:

```
protected void LoginAction_Click(object sender, EventArgs e)
{
    Page.Validate();
    if (!Page.IsValid) return;

    if (FormsAuthentication.Authenticate(UsernameText.Text, PasswordText.Text))
    {
        // Create the authentication cookie
        HttpCookie AuthCookie;
        AuthCookie = FormsAuthentication.GetAuthCookie(
            UsernameText.Text, true);
        AuthCookie.Expires = DateTime.Now.AddDays(10);

        // Add the cookie to the response
        Response.Cookies.Add(AuthCookie);

        // Redirect to the originally requested page
        Response.Redirect(FormsAuthentication.GetRedirectUrl(
            UsernameText.Text, true));
    }
    else
    {
        // User name and password are not correct
        LegendStatus.Text = "Invalid username or password!";
    }
}
```

The code for checking the credentials is the same in this scenario. The only difference is that the authentication cookie isn't added automatically. Instead, it's created with a call to `GetAuthCookie()`, which returns a new instance of `HttpCookie`, as shown here:

```
HttpCookie AuthCookie;
AuthCookie = FormsAuthentication.GetAuthCookie(
    UsernameText.Text, true);
```

Once you've created the authentication cookie, you can retrieve the current date and time (using the `DateTime.Now` static property), add ten days to it (using the `DateTime.AddDays()` method), and use this value as the expiry date and time of the cookie:

```
AuthCookie.Expires = DateTime.Now.AddDays(10);
```

Next, you have to add the cookie to the HTTP response:

```
Response.Cookies.Add(AuthCookie);
```

Finally, you can redirect the user to the originally requested URL, which you can obtain by using the `GetRedirectUrl()` method:

```
Response.Redirect(FormsAuthentication.GetRedirectUrl(
    UsernameText.Text, true));
```

The end result is a cookie that will persist beyond the closing of the browser but that will expire after ten days, at which point the user will need to reenter credentials to log into the website.

Summary

In this chapter, you learned how to use forms authentication to implement authentication systems that simplify life and provide a great deal of flexibility. You also learned how to protect passwords and how you can use any data source for credential storage. In the next chapter, you'll learn about the new features that are built on top of forms authentication and that make it even easier to create login pages and deal with user authentication without writing all the code yourself.

