Sitecore CMS 6.0 and later

Security API Cookbook

A Conceptual Overview for CMS Developers

Table of Contents

Chapter 1

Introduction

This document provides sample code to introduce Sitecore APIs that support common security requirements, such as user management, authentication, authorization, and user profile management. This document provides an overview of concepts rather than describing every API used. Sitecore provides detailed descriptions of API methods in the API documentation.[1] For the reader's convenience, this document describes some security APIs provided by the ASP.NET framework, but not abstracted in any way by Sitecore.[2]

This document contains the following chapters:

- Chapter 1 – Introduction
- Chapter 2 – User, Domain, Role, and Profile Management
- Chapter 3 – User Profiles
- Chapter 4 – Access Rights Management
- Chapter 5 – System.Web.Security APIs
- Chapter 6 – Appendix A

[1] For access to the Sitecore API documentation, see http://sdn.sitecore.net/Reference/Sitecore%206/Sitecore_6_API_Reference.aspx.
[2] For more information about the System.Web.Security APIs, see http://msdn.microsoft.com/en-us/library/system.web.security.aspx.

Chapter 2

User, Domain, Role, and Profile Management

This chapter describes users, domains, roles, and user profiles, and provides sample code to manage roles and users, including authentication (login), self-registration, and password management. For more information about user profiles, see Chapter 3, User Profiles.

This chapter contains the following sections:

- Sitecore Security Overview
- Sitecore Security API Overview
- Membership Provider
- Sample Login Form
- Sample Self-Registration Form
- Sample Password Recovery Form
- Virtual Users

2.1 Sitecore Security Overview

A Sitecore user represents an individual that accesses the system.

Each user has a profile, which defines user properties such as full name and email address. Sitecore provides a default profile for all users. You can also implement custom user profiles to manage custom user properties more easily.

A role is a collection of users and nested roles. Each user can be a member of any number of roles. A user has the access rights associated with each of their roles, including nested roles.

Each role can be a member of any number of roles. Sitecore refers to a role that contains other roles as a target role, and the contained roles as nested roles. A nested role has the access rights associated with each of the target roles that contain the role in the same way that a user has the access rights associated with each of their roles. A user has the access rights associated with each of the target roles containing each of the nested roles of which they are a member.

A Sitecore domain is a collection of users and roles. Most users and roles (accounts) exist within a domain. A Sitecore solution can include any number of domains.

An account name consists of a domain and a local user name separated by a backslash character ("\"). For example, in the username `domain\username`, the `domain` part represents the domain name, and the `username` represents the local user name. All references to usernames should include the domain.

Sitecore ships configured with two domains: the Sitecore domain containing information about CMS users and roles, and the Extranet domain containing information about users and roles of the published Web sites.

Sitecore associates a context domain with each logical Web site. For example, the context domain associated with the default published Web site is the Extranet domain, while the context domain associated with the CMS user interfaces is the Sitecore domain.

All code that accesses Sitecore APIs runs in the context of a Sitecore user. Without calling Sitecore APIs to authenticate, code runs in the context of the Anonymous user. While the Anonymous user acts as an individual user, it represents the entire class of users who have not authenticated.

If code accesses Sitecore APIs without authenticating, that code runs in the context of the Anonymous user on the Extranet domain. Because you cannot access the CMS without authenticating as a user in the Sitecore domain, code that accesses Sitecore APIs from within the CMS user interfaces runs as a specific user in the Sitecore domain.

Note
The Sitecore Extranet domain can represent any published Web site that involves authentication, which may be more inclusive than other definitions of the term extranet. You do not have to implement additional domains to enable authentication on the published Web sites.

Important
Unless otherwise specified, all strings containing account names must include the domain name. For example, in the full account name `domain\account`, `domain` represents the domain name and `account` represents the account name.

Important
Domains contain account definitions, not access right settings. In order for access right changes to appear in publishing target databases, you must publish the affected items.

2.2 Sitecore Security API Overview

The Sitecore security model abstracts some features of the underlying .NET membership, role, and profile providers.[3] In many cases, you can achieve the same functionality using Sitecore API methods or methods directly in the .NET framework. For example, you can develop custom solutions that invoke Sitecore APIs to authenticate users, or you can use ASP.NET membership controls with Sitecore solutions.[4]

Sitecore does not abstract all functions provided by the underlying .NET security model. The methods and controls provided by the underlying .NET framework may be slightly more efficient than the Sitecore API methods, such as those that accept strings instead of objects. In cases where you have already created the object, the Sitecore API methods may be more convenient, and the performance differential should be negligible.

Sitecore APIs related to security include the following:[5]

- **`Sitecore.Context`**: Contains properties that indicate the context user and whether that user has authenticated.
- **`Sitecore.Security.Domains.Domain`**: Represents a Sitecore security domain. A security domain contains user and role definitions, including user profiles and passwords (encrypted by default). Sitecore security domains function much like Windows or Active Directory domains. Sitecore supports multiple domains. The default security domains include the Sitecore domain containing CMS users and roles, and the Extranet domain containing Web site users and roles. Each account is a member of a domain.
- **`Sitecore.Security.Accounts.Account`**: Is the base class for `Sitecore.Security.Accounts.Role` and `Sitecore.Security.Accounts.User`.
- **`Sitecore.Security.Accounts.Role`**: Represents a named collection of users and member roles.
- **`Sitecore.Security.Accounts.UserRoles`**: Represents a collection of roles associated with a user.
- **`Sitecore.Security.Accounts.RolesInRolesManager`**: Provides methods for working with nested roles.
- **`Sitecore.Security.Accounts.User`**: Represents a named user.
- **`Sitecore.Security.UserProfile`**: Represents the profile of a named user.
- **`Sitecore.Security.Authentication.AuthenticationManager`**: Provides methods for authentication.
- **`Sitecore.Security.Authentication.AuthenticationHelper`**: Provides additional methods for authentication.
- **`Sitecore.Security.Accounts.UserSwitcher`**: Causes code to run in the context of a different user.
- **`Sitecore.SecurityModel.SecurityDisabler`**: Causes code to run in the context of a user with administrative rights.
- **`Sitecore.Security.AccessControl.AuthorizationManager`**: Contains methods to determine and apply access rights.

[3] For more information about the .NET provider model, see http://msdn.microsoft.com/en-us/library/tw292whz.aspx and http://msdn.microsoft.com/en-us/library/aa479030.aspx.

[4] For more information about the ASP.NET membership controls, see http://msdn.microsoft.com/en-us/library/ms178329.aspx.

[5] For access to the Sitecore API documentation, see http://sdn.sitecore.net/Reference/Sitecore%206/Sitecore_6_API_Reference.aspx.

- **`Sitecore.Security.AccessControl.AccessRight`**: Represents an access right.
- **`Sitecore.Security.AccessControl.AccessPermission`**: Represents an access right permission state.
- **`Sitecore.Security.AccessControl.PropagationType`**: Represents a rule for applying an access right to descendants of an item.
- **`Sitecore.Security.AccessControl.AccessRuleCollection`**: Represents a collection of access rules for an item.
- **`Sitecore.Security.AccessControl.AccessRuleCollectionHelper`**: Provides methods for working with access rights.

Note
Sitecore does not abstract every security feature provided by the underlying ASP.NET membership, role, and profile providers. For more information about relevant methods in the ASP.NET provider framework, see Chapter 5, System.Web.Security APIs.[6]

[6] For more information about the System.Web.Security APIs, see http://msdn.microsoft.com/en-us/library/system.web.security.aspx.

2.3 Membership Provider Configuration

Sitecore security relies on ASP.NET membership, role, and profile providers.[7]

Note
Because Sitecore uses ASP.NET membership, role, and profile providers, you can use ASP.NET login controls instead of writing code to invoke Sitecore APIs.[8] This document references specific ASP.NET login controls where appropriate.

You can configure the ASP.NET membership provider, including options for password reset and retrieval, using the appropriate `/configuration/system.web/membership/providers/add` element in `web.config`.[9] Update the `<add>` element that has a `name` attribute that is equal to the value of the `realProviderName` attribute of the `<add>` element with `name sitecore`. For Microsoft SQL server installations, `realProviderName` is `sql`; update the `<add>` element with `name` attribute `sql`. You can specify the following attributes in this element:

- **`enablePasswordReset`** (`true` or `false`): Whether passwords can be reset.
- **`enablePasswordRetrieval`** (`true` or `false`): Whether passwords can be retrieved. Requires `passwordFormat` of `clear`.
- **`maxInvalidPasswordAttempts`** (positive integer): Number of invalid password attempts to accept before locking a user out of the system.
- **`minRequiredNonalphanumericCharacters`** (`0` or positive integer): Number of special characters required.
- **`minRequiredPasswordLength`** (positive integer): Minimum allowed password length.
- **`passwordFormat`** (`Clear`, `Encrypted`, or `Hashed`): Password storage format.[10]
- **`passwordStrengthRegularExpression`** (`string`): Password must match regular expression specified.
- **`requiresQuestionAndAnswer`** (`true` or `false`): Password reset requires correct answer to question stored in user's profile.
- **`requiresUniqueEmail`** (`true` or `false`): Whether each user must have a unique email address.

Note
If `passwordFormat` is `Encrypted`, .NET uses encryption keys in `machine.config`, which should be configured with the same values on all systems that must authenticate users using these encrypted passwords.

The default Sitecore configuration does not support password recovery if a user cannot provide their password. You can use `System.Web.Security.MembershipUser.ResetPassword()` method

[7] For more information about Sitecore's use of ASP.NET membership providers, see http://sdn.sitecore.net/Articles/Security/Low_level_Sitecore_Security_and_Custom_Providers.aspx. For more information about ASP.NET membership providers, see http://msdn.microsoft.com/en-us/library/aa479031.aspx. For more information about ASP.NET role providers, see http://msdn.microsoft.com/en-us/library/aa479032.aspx. For more information about ASP.NET profile providers, see http://msdn.microsoft.com/en-us/library/aa479035.aspx.
[8] For more information about the ASP.NET login controls, see http://msdn.microsoft.com/en-us/library/ms178329.aspx.
[9] For more information about ASP.NET membership provider configuration attributes, see http://msdn.microsoft.com/en-us/library/whae3t94.aspx.
[10] For more information about ASP.NET password storage formats, see http://msdn.microsoft.com/en-us/library/system.web.security.membershipprovider.passwordformat.aspx.

to reset the user's password to a temporary password consisting of a random string, and make that password available to the user. The user can then log in using this random password, and then change their password, without ever recovering their lost password.

2.4 Sample Login Form

You can implement a sublayout to contain a login form with code-behind to authenticate users, by default using the Extranet domain.

Note
As an alternative to invoking APIs directly, you can authenticate published web site users using the ASP.NET Login Web control. [11] For more information about the ASP.NET Login Web control, see the following section How to Use the ASP.NET Login Web Control.

The following sample code for a sublayout file implements a very simple login data entry form:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Login.ascx.cs"
  Inherits="Namespace.Web.UI.Login" %>

Username: <asp:textbox id="txtUsername" runat="server" /><br />
Password: <asp:textbox id="txtPassword" runat="server" textmode="password" /><br />
Persistent: <asp:checkbox id="chkPersist" runat="server" /><br />
<asp:button id="btnGo" runat="server" Text="Go" /><br />
<asp:label id="lblMessage" runat="server" />
```

This login form consists of:

- A text field for the user to enter their user name.
- A password field for the user to enter their password.
- A checkbox controlling whether authentication is persistent or applies only to the session.
- A button to submit the form.
- A label to contain any error message that results from submitting the form.

Note
Authentication applies to the session by default and is not persistent. Session authentication creates a session cookie. With session authentication, if the user closes the browser window, they must re-enter a username and password to authenticate when they return to the Web site. Persistent authentication creates a browser cookie. With persistent authentication, the user does not need to re-enter a username and password when they return to the Web site.

The following sample code for a sublayout code-behind file implements logic to authenticate a user:

```
using System;

namespace Namespace.Web.UI
{
  public partial class Login : System.Web.UI.UserControl
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if (IsPostBack)
      {
        if (String.IsNullOrEmpty(txtUsername.Text))
        {
          lblMessage.Text = "Invalid username.";
        }
        else if (String.IsNullOrEmpty(txtPassword.Text))
        {
          lblMessage.Text = "Invalid password.";
        }
        else
        {
          try
          {
```

[11] For more information about the ASP.NET Login Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.login.aspx.

```
                    Sitecore.Security.Domains.Domain domain = Sitecore.Context.Domain;
                    string domainUser = domain.Name + @"\" + txtUsername.Text;

                    if
(Sitecore.Security.Authentication.AuthenticationManager.Login(domainUser,
                      txtPassword.Text, chkPersist.Checked))
                    {
                      Sitecore.Web.WebUtil.Redirect("/");
                    }
                    else
                    {
                      throw new System.Security.Authentication.AuthenticationException(
                        "Invalid username or password.");
                    }
                  }
                  catch (System.Security.Authentication.AuthenticationException)
                  {
                    lblMessage.Text = "Processing error.";
                  }
                }
              }
            }
          }
        }
```

The logic behind this code is as follows:

1. If the page is not posting back, then the user has not had a chance to enter data into the form. In this case, do nothing.
2. If the user has not entered a username, then display an error message, and do nothing else.
3. If the user has not entered a password, then display an error message, and do nothing else.
4. Determine the full username from the context domain and the user name entered by the user.
5. If the system can authenticate the user using the full username and the password entered by the user, then redirect to the home page, and do nothing else.
6. If the system cannot authenticate the user, throw an exception, which will display a generic error message, and do nothing else.

Important
Note the exception management logic in the preceding code. This code outputs a generic error message regardless of the cause of the exception. Never output a message that indicates specifically that a username or password is invalid, as this could allow attackers to harvest usernames. Consider SSL encryption for login forms, or at least encryption of passwords sent over HTTP. Consider using a third factor for authentication.

Note
To log a user out, call `Sitecore.Security.Authentication.AuthenticationManager.Logout()`.

2.4.1 How to Use the ASP.NET Login Web Control

You can authenticate users using the ASP.NET Login Web control.[12] You must consider the domain containing the users when using this control. For example, add an ASP.NET Login Web control to a sublayout:

```
        <asp:login id="loginControl" runat="server" DestinationPageUrl="/" />
```

Add code-behind to the sublayout based on the following to provide logic to add the context domain to the username, but remove it if the password entered by the user is invalid, so that the user never sees the domain name:

[12] For more information about the ASP.NET Login Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.login.aspx.

```
using System;

namespace Namespace.Web.UI
{
  public partial class LoginForm : System.Web.UI.UserControl
  {
    private string _usernameAsEntered = String.Empty;

    protected override void OnInit(EventArgs e)
    {
      base.OnInit(e);
      loginControl.LoggingIn += new LoginCancelEventHandler(this.Login_LoggingIn);
      loginControl.LoginError += new EventHandler(this.Login_LoginError);
    }

    private void Login_LoggingIn(object sender, LoginCancelEventArgs e)
    {
      string domainUser = Sitecore.Context.Domain.GetFullName(loginControl.UserName);

      if (System.Web.Security.Membership.GetUser(domainUser) != null)
      {
        _usernameAsEntered = loginControl.UserName;
        loginControl.UserName = domainUser;
      }
    }

    private void Login_LoginError(object sender, EventArgs e)
    {
      loginControl.UserName =  usernameAsEntered;
    }
  }
}
```

The logic behind this code is as follows:

1. When the user submits the ASP.NET Login control, determine the full username by adding the context domain name and the backslash character to the user name entered by the user. If that user exists, store the original user name entered by the user, and set the value of the user name input field to the full username including the domain. This provides the domain name to the ASP.NET Login Web control logic that authenticates the user.
2. If the system is unable to authenticate the user, then reset the value of the user name input field to the value entered by the user. This removes the context domain, so that the user is never aware of the domain.

2.5 Sample Self-Registration Form

Many Web sites allow users to register, which associates the user with a username, password, and user profile properties such as role membership. Registration typically provides the user with access to additional content or other features. You can implement a sublayout to contain a self-registration form with code-behind to register new Web site users.

Note

3. As an alternative to invoking APIs directly, you can authenticate users with an ASP.NET CreateUserWizard Web control.[13] For more information about using the ASP.NET CreateUserWizard Web control, see the following section Log the user in. If the system authenticated the user, then redirect the user /profile.aspx to manage their profile. If the system did not authenticate the user, then throw an exception, resulting in a generic error message.

Using the ASP.NET CreateUserWizard Web Control.

Note
Call the `Sitecore.Security.Accounts.User.Delete()` method to remove a user. For example, to delete the user `user` in the domain `domain`:

```
string domainUser = @"domain\user";

if (Sitecore.Security.Accounts.User.Exists(domainUser))
{
  Sitecore.Security.Accounts.User user =
    Sitecore.Security.Accounts.User.FromName(domainUser,false);
  user.Delete();
}
```

The following sample code for a sublayout file implements a very simple self-registration data entry form:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Register.ascx.cs"
  Inherits="Namespace.Web.UI.Register" %>

Username: <asp:textbox id="txtUsername" runat="server" /><br />
Email: <asp:textbox id="txtEmail" runat="server" /><br />
Password: <asp:textbox id="txtPassword" textmode="password" runat="server" /><br />
Confirm: <asp:textbox id="txtPasswordConfirm" textmode="password" runat="server" /><br
/>
Question: <asp:textbox id="txtQuestion" textmode="password" runat="server" /><br />
Answer:<asp:textbox id="txtAnswer" TextMode="password" runat="server" /><br />
Persistent: <asp:checkbox id="chkPersist" runat="server" /><br />
<asp:Button id="btnGo" text="Go" runat="server" /><br />
<asp:Label id="lblMessage" runat="server" />
```

This self-registration form contains:

- A text field for the user to enter their desired user name.
- A text field for the user to enter their email address.
- A password field for the user to enter their password.
- A password field for the user to confirm that password, to help ensure they did not enter their password erroneously.
- A text field for the user to enter a security profile question.
- A text field for the user to enter an answer to that security question.

[13] For more information about the ASP.NET CreateUserWizard Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.createuserwizard.aspx.

- A checkbox for the user to control whether authentication should be persistent or apply to a session.
- A button to submit the form.
- A label to contain any error message that results from submitting the form.

Note
In most implementations, the user does not enter the text of the security question. The user typically answers one predefined question, or selects one or more questions from one or more lists provided by the system. When possible, store the ID of the question rather than its text. For multiple questions and answers, store XML or a pipe-separated list. Store answers in a similar format.

The following sample code for a sublayout code-behind file implements logic to create a user:

```
        using System;

        namespace Namespace.Web.UI
        {
          public partial class Register : System.Web.UI.UserControl
          {
            protected void Page Load(object sender, EventArgs e)
            {
              if (IsPostBack)
              {
                this.lblMessage.Text = String.Empty;

                if(String.IsNullOrEmpty(txtUsername.Text))
                {
                  lblMessage.Text = "Invalid username.";
                }
                else if(System.Web.Security.Membership.Provider.RequiresUniqueEmail
                  && String.IsNullOrEmpty((txtEmail.Text)))
                {
                  lblMessage.Text = "Invalid email address.";
                }
                else if(System.Web.Security.Membership.RequiresQuestionAndAnswer
                  && (String.IsNullOrEmpty(txtQuestion.Text)
                  || String.IsNullOrEmpty(txtAnswer.Text)))
                {
                  lblMessage.Text = "Specify question and answer.";
                }
                else if(String.IsNullOrEmpty(txtPassword.Text)
                  || String.IsNullOrEmpty(txtPasswordConfirm.Text))
                {
                  lblMessage.Text = "Invalid password.";
                }
                else if(txtPassword.Text!=txtPasswordConfirm.Text)
                {
                  lblMessage.Text = "Passwords don't match.";
                }
                else
                {
                  string domainUser = Sitecore.Context.Domain.GetFullName(txtUsername.Text);

                  try
                  {
                    if(Sitecore.Security.Accounts.User.Exists(domainUser))
                    {
                      throw new System.Web.Security.MembershipCreateUserException(
                        domainUser + " exists.");
                    }
                    else if(System.Web.Security.Membership.Provider.RequiresUniqueEmail
                      && !String.IsNullOrEmpty(

System.Web.Security.Membership.Provider.GetUserNameByEmail(txtEmail.Text)))
                    {
                      throw new System.Web.Security.MembershipCreateUserException(
                        txtEmail.Text + " already registered.");
                    }
                    else
                    {
```

```
                    System.Web.Security.MembershipCreateStatus status;
                    System.Web.Security.Membership.CreateUser(domainUser,
                      txtPassword.Text, txtEmail.Text, txtQuestion.Text,
                      txtAnswer.Text, true, out status);

                    if(!status.Equals(System.Web.Security.MembershipCreateStatus.Success))
                    {
                      throw new System.Web.Security.MembershipCreateUserException(
                        status.ToString());
                    }

                    if(Sitecore.Security.Authentication.AuthenticationManager.Login(
                      domainUser, txtPassword.Text, chkPersist.Checked))
                    {
                      Sitecore.Web.WebUtil.Redirect("/profile.aspx");
                    }
                    else
                    {
                      throw new System.Web.Security.MembershipCreateUserException(
                        "Unable to login after creating " + domainUser );
                    }
                  }
                }
                catch(System.Web.Security.MembershipCreateUserException)
                {
                  lblMessage.Text = "Processing error.";
                }
              }
            }
          }
        }
      }
```

The logic behind this code is as follows:

1. If the page is not posting back, then the user has not had a chance to enter data into the form. In this case, do nothing.
2. If the user has not entered a username, then display an error message, and do nothing else.
3. If system configuration requires that each user have a unique email address, and the user has not entered an email address, then display an error message and do nothing else.
4. If system configuration requires that each user specify a security question and answer, and the user has not entered one or both of these fields, then display an error message and do nothing else.
5. If the user has not specified a password or confirmed their password, then display an error message and do nothing else. If the two passwords entered by the user are not equal, then display an error message and do nothing else.
6. Determine the username from the context domain and the user name entered by the user. If the username already exists, then throw an exception, which will display a generic error message, and do nothing else.
7. If system configuration requires that each user have a unique email address, and the user has not entered a unique email address, then throw an exception, which will display a generic error message, and do nothing else.
8. Attempt to create the user. If the system could not create the user, then throw an exception, which will display a generic error message, and do nothing else.
9. Log the user in. If the system authenticated the user, then redirect the user /profile.aspx to manage their profile. If the system did not authenticate the user, then throw an exception, resulting in a generic error message.

2.5.1 Using the ASP.NET CreateUserWizard Web Control

You can create users using the ASP.NET CreateUserWizard Web control instead of writing code-behind.[14] You must consider the domain containing the users when using this control.

For example, add an ASP.NET CreateUserWizard Web control to a sublayout:

```
<asp:CreateUserWizard runat="server" id="createUserWizardControl"
  ContinueDestinationPageUrl="/" />
```

Add code-behind to the sublayout based on the following to provide logic to add the context domain to the username so that the user does not have to specify it, but remove it if Sitecore cannot create the user:

```
using System;

namespace Namespace.Web.UI
{
  public partial class CreateUser : System.Web.UI.UserControl
  {
    private string  usernameAsEntered = String.Empty;

    private void CreateUserWizard_CreatingUser(object sender, EventArgs e)
    {
      string domainUser =
        Sitecore.Context.Domain.GetFullName(createUserWizardControl.UserName);

      if (System.Web.Security.Membership.GetUser(domainUser) == null)
      {
        _usernameAsEntered = createUserWizardControl.UserName;
        createUserWizardControl.UserName = domainUser;
      }
    }

    private void CreateUserWizard_CreateUserError(object sender, EventArgs e)
    {
      createUserWizardControl.UserName =  usernameAsEntered;
    }

    protected override void OnInit(EventArgs e)
    {
      base.OnInit(e);
      createUserWizardControl.CreatingUser +=
        new LoginCancelEventHandler(this.CreateUserWizard CreatingUser);
      createUserWizardControl.CreateUserError +=
        new CreateUserErrorEventHandler(this.CreateUserWizard_CreateUserError);
    }
  }
}
```

The logic behind this code is as follows:

1. When the user submits the ASP.NET CreateUserWizard control, determine the full username by adding the context domain name and the backslash character to the user name entered by the user. If that user exists, store the original user name entered by the user, and set the value of the user name input field to the full username including the domain. This provides the domain name to the ASP.NET CreateUserWizard Web control logic that creates the user.
2. If the system is unable to create the user, then reset the value of the user name input field to the value entered by the user. This removes the context domain, so that the user is never aware of the domain.

[14] For more information about the ASP.NET CreateUserWizard Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.createuserwizard.aspx.

2.6 Sample Password Recovery Form

You can implement a sublayout to contain a form with code-behind to allow users to recover or reset their passwords.

Note
As an alternative to invoking APIs directly, you can authenticate users with an ASP.NET PasswordRecovery Web control.[15]

Important
The password reset or recovery form must be accessible to unauthenticated users.

The following sample code for a sublayout file implements a very simple password recovery data entry form:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="LostPassword.ascx.cs"
  Inherits="Namespace.Web.UI.LostPassword" %>

Username: <asp:textbox id="txtUsername" runat="server" /><br />
Answer: <asp:textbox id="txtAnswer" runat="server" textmode="password" /><br />
<asp:button id="btnGo" runat="server" text="Go" /><br />
<asp:label id="lblMessage" runat="server" /><br />
```

This login form consists of:

- A text field for the user to enter their user name.
- A password field for the user to enter the answer to the security question associated with their account.
- A checkbox controlling whether authentication is persistent or applies only to the session.
- A button to submit the form.
- A label to contain any error message that results from submitting the form.

The following sample code for a sublayout code-behind file implements logic to recover or reset a user's password:

```
using System;

namespace Namespace.Web.UI
{
  public partial class LostPassword : System.Web.UI.UserControl
  {
    protected void Page Load(object sender, EventArgs e)
    {
      if(IsPostBack)
      {
        lblMessage.Text = String.Empty;

        if(String.IsNullOrEmpty(txtUsername.Text))
        {
          lblMessage.Text = "Invalid user.";
        }
        else if(System.Web.Security.Membership.RequiresQuestionAndAnswer
          && String.IsNullOrEmpty(txtAnswer.Text))
        {
          lblMessage.Text = "Invalid answer.";
        }
        else
        {
          try
```

[15] For more information about the ASP.NET PasswordRecovery Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.passwordrecovery.aspx and http://msdn.microsoft.com/en-us/library/ms178335.aspx.

```
                {
                  string domainUser = Sitecore.Context.Domain.GetFullName(txtUsername.Text);

                  if(!Sitecore.Security.Accounts.User.Exists(domainUser))
                  {
                    throw new System.Security.Authentication.AuthenticationException(
                      domainUser + " does not exist.");
                  }
                  else
                  {
                    System.Web.Security.MembershipUser user =
                      System.Web.Security.Membership.GetUser(domainUser);

                    if(System.Web.Security.Membership.EnablePasswordRetrieval)
                    {
                      lblMessage.Text = "Password for " + user.UserName + ": ";

                      if(System.Web.Security.Membership.RequiresQuestionAndAnswer)
                      {
                        lblMessage.Text += user.GetPassword(txtAnswer.Text);
                      }
                      else
                      {
                        lblMessage.Text += user.GetPassword();
                      }
                    }
                    else if(System.Web.Security.Membership.EnablePasswordReset)
                    {
                      lblMessage.Text = "New password for " + user.UserName + ": ";

                      if(System.Web.Security.Membership.RequiresQuestionAndAnswer)
                      {
                        lblMessage.Text += user.ResetPassword(txtAnswer.Text);
                      }
                      else
                      {
                        lblMessage.Text += user.ResetPassword();
                      }
                    }
                    else
                    {
                      throw new System.Configuration.ConfigurationErrorsException(
                        "Cannot retrieve or reset passwords.");
                    }
                  }
                }
                catch(System.Security.Authentication.AuthenticationException)
                {
                  lblMessage.Text = "Processing error.";
                }
                catch(System.Configuration.ConfigurationErrorsException)
                {
                  lblMessage.Text = "Configuration error.";
                }
              }
            }
          }
        }
      }
```

The logic behind this code is as follows:

1. If the page is not posting back, then the user has not had a chance to enter data into the form. In this case, do nothing.
2. Clear the label that may contain an error message from a previous submission of the form with invalid data.
3. If the user has not entered a user name, then set an error message, and do nothing else.
4. If system configuration requires an answer to the security question associated with the user and the user has not entered an answer, then set an error message, and do nothing else.
5. Determine the username from the context domain and the user name entered by the user.

6. If the account specified by the user does not exist, then throw an exception, which will display a generic error message, and do nothing else.
7. If system configuration allows password retrieval, then display the user's password, and do nothing else.
8. If system configuration allows passwords to be reset, then reset the user's password, display that new password, and do nothing else.
9. Throw an exception indicating invalid configuration.

Note
If a user attempts to authenticate using an invalid password more than the number of times allowed by the `maxInvalidPasswordAttempts` attribute of the appropriate membership provider defined in `web.config`, the provider will lock the user out of the system. The user may experience lockout as a lost password condition. For information about unlocking a user, see the section System.Web.Security.MembershipUser.UnlockUser().

2.7 Virtual Users

Virtual users allow you to integrate third-party authentication systems without the need to implement a custom ASP.NET membership provider. Virtual users provide authentication through the third-party system, but often use Sitecore's role provider for authorization. In this way, you can manage users in a central repository, but CMS audience segments and other authorization roles using Sitecore security.

Important
Virtual users are transitory; they do not persist on the system after the user logs out. Virtual users are like other users in all other respects except as described in this section.

For example, you could implement a login form for virtual users similar to the login form example in the section Sample Login Form. Instead of authenticating the user against Sitecore, authenticate them against your existing security system. Then call Sitecore APIs as described below to authenticate the virtual user. You can associate the virtual user with Sitecore roles and control their access to data in the CMS database using access rights without implementing a custom membership or role provider.

2.7.1 How to Create a Virtual User

The `Sitecore.Security.Authentication.AuthenticationManager.BuildVirtualUser()` method builds and returns a virtual user with the username specified by the first parameter. The second parameter controls whether the virtual user is authenticated. For example, to build the virtual user `user` in the domain `domain` with the full name `full name` as a member of the role `role` in the domain `domain`, and log in that user:

```
        Sitecore.Security.Accounts.User user =
        Sitecore.Security.Authentication.AuthenticationManager.BuildVirtualUser(@"domain\user"
, true);

        if (user != null)
        {
          string domainRole = @"domain\role";
          if (Sitecore.Security.Accounts.Role.Exists(domainRole))
          {
            user.Roles.Add(Role.FromName(domainRole));
          }

          user.Profile.Email = "user@domain.com";
          user.Profile["Custom Property"] = "Custom Value";
          user.Profile.Save();

          Sitecore.Security.Authentication.AuthenticationManager.LoginVirtualUser(user);
        }
```

Important
You must log in a virtual user only after you assign *Roles and Profile* properties to them. The *Roles and Profile* properties that are assigned after logging in are lost upon subsequent request.

When you work with a virtual user, you must only use the `AuthenticationManager.LoginVirtualUser(User user)` or `AuthenticationManager.Login(User user)` method. You must log in virtual users with a User object rather than a user name. If you log in virtual users with a User object, you do not have to create a physical record for each virtual user in *Core* database.

2.8 Sitecore.Security.Accounts.RolesInRolesManager APIs

This class describes how .NET APIs operate with roles in roles, namely how they add roles to roles, remove roles from roles, and get roles from a role.

2.8.1 How to Add a Role to a Role

You can add an existing role to another existing role using the `Sitecore.Security.Accounts.RolesInRolesManager.AddRoleToRole (Role memberRoles, Role targetRoles)` function. The first parameter is a member role and the second is a target role.

For example, to assign a role to another role:

```
        const string parentRole = @"sitecore\Author";
        const string memberRole = @"sitecore\MyRole";
```

```
        if (RolesInRolesManager.RolesInRolesSupported &&
!RolesInRolesManager.IsRoleInRole(Role.FromName(memberRole), Role.FromName(parentRole),
false))
            {
               RolesInRolesManager.AddRoleToRole(Role.FromName(memberRole),
Role.FromName(parentRole));
            }
```

2.8.2 How to Remove a Role from a Role

You can remove an existing role from another existing role using the `Sitecore.Security.Accounts.RolesInRolesManager.RemoveRoleFromRole (Role memberRoles, Role targetRoles)` function. The first parameter is a member role and the second one is a target role.

For example to remove a role from another role:

```
        const string parentRole = @"sitecore\Author";
        const string memberRole = @"sitecore\MyRole";
```

```
        if (RolesInRolesManager.RolesInRolesSupported &&
RolesInRolesManager.IsRoleInRole(Role.FromName(memberRole), Role.FromName(parentRole), false))
                    {
                        RolesInRolesManager.RemoveRoleFromRole(Role.FromName(memberRole),
Role.FromName(parentRole));
                    }
```

2.8.3 How to Get a Role from a Role

You can get an existing role from another existing role using the `Sitecore.Security.Accounts.RolesInRolesManager.GetRolesInRole (Role targetRole, bool includeIndirectMembership)` function. The first parameter is a role and the second parameter determines whether you need indirect membership.

For example, to get roles from another role:

```
        const string parentRole = @"sitecore\Author";

        IEnumerable<Role> roleList =
RolesInRolesManager.GetRolesInRole(Role.FromName(parentRole), false);
```

Chapter 3

User Profiles

This chapter describes solutions and provides sample code to customize user profiles.

This chapter contains the following sections:

- Overview of User Profiles
- How to Access Standard User Profile Properties
- How to Access Custom User Profile Properties
- How to Extend the Default User Profile
- Implement a Custom User Profile
- Sample User Profile Management Form

3.1 Overview of User Profiles

Sitecore associates each user with a user profile. The default user profile contains properties such as the full name and email address of the user. Sitecore provides APIs to access these specific user profile properties, as well as APIs to access custom profile properties.

You can extend the default user profile or implement a custom user profile to include fields for manipulating custom user profile properties in the User Manager. Extend the default user profile or implement a custom user profile to provide the ability to manipulate custom user profile properties through the User Manager.

You can implement a .NET class to standardize programmatic access to custom user profile properties.

It is not necessary to extend the default profile, to implement a custom user profile, to use a custom class to represent user profiles, or to implement an ASP.NET profile provider, in order to access custom profile properties. You can simply use the methods described in the following section How to Access Custom User Profile Properties.

3.2 How to Access Standard User Profile Properties

By default, Sitecore uses `Sitecore.Security.UserProfile` to represent user profiles. This class contains the following properties:

- **`FullName`**: The full name of the user.
- **`Email`**: The email address of the user.
- **`Comment`**: A comment associated with the user.
- **`Portrait`**: The URL of an image associated with user.

You can access a user's profile through the `Sitecore.Security.Accounts.User.Profile` property. For example, to access the email address of the context user:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
Sitecore.Security.UserProfile profile = user.Profile;
string userEmail = profile.Email;
```

Important
You must call the `Sitecore.Security.UserProfile.Save()` method after setting a user profile property. For example:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
Sitecore.Security.UserProfile profile = user.Profile;
profile.Email = "address@domain.tld";
profile.Save();
```

Note
You cannot set profile properties for a user that is not authenticated. To retrieve an authenticated user, pass True as the second parameter to `Sitecore.Security.Accounts.User.FromName()`. For example, to retrieve the authenticated user `user` in the domain `domain`:

```
Sitecore.Security.Accounts.User user =
  Sitecore.Security.Accounts.User.FromName(@"domain\user",true);
```

3.3 How to Access Custom User Profile Properties

The `Sitecore.Security.UserProfile` class provides methods to set, retrieve, and remove custom user profile properties.

The `Sitecore.Security.UserProfile.GetCustomPropertyNames()` method returns the names of the custom properties in the user's profile.

Important
The `Sitecore.Security.UserProfile.GetCustomPropertyNames()` method returns only those property names that defined for the user. This may not be the complete list of all custom property names used by the solution.

The `Sitecore.Security.UserProfile.GetCustomProperty()` method returns the value of the custom profile property specified by the first parameter. For example, to output the names and values for all custom properties defined for the context user:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
Sitecore.Security.UserProfile profile = user.Profile;

foreach(string attributeKey in profile.GetCustomPropertyNames())
{
  string attributeValue = profile.GetCustomProperty(attributeKey);
  //TODO: handle attributeKey and attributeValue
}
```

Note
As an alternative to the `Sitecore.Security.UserProfile.GetCustomProperty()` method, you can access custom user profile properties through the collection exposed by the `Sitecore.Security.UserProfile` class. For example:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
Sitecore.Security.UserProfile profile = user.Profile;
string attributeValue = profile[attributeKey];
```

The `Sitecore.Security.UserProfile.SetCustomProperty()` method sets the custom profile property specified by the first parameter to the value specified by the second parameter. You cannot set custom user profile properties for a user that is not authenticated. You must invoke `Sitecore.Security.UserProfile.Save()` after calling this method. For example, to set the custom property named `attributeKey` to the value `attributeValue` in the context user's profile:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
Sitecore.Security.UserProfile profile = user.Profile;
profile.SetCustomProperty("attributeKey ","attributeValue");
profile.Save();
```

Note
As an alternative to the `Sitecore.Security.UserProfile.SetCustomProperty()` method, you can set custom properties through the collection exposed by the `Sitecore.Security.UserProfile` class. For example:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
Sitecore.Security.UserProfile profile = user.Profile;
profile["attributeKey"] = "attributeValue";
profile.Save();
```

The `Sitecore.Security.UserProfile.RemoveCustomProperty()` method removes a custom property from a user's profile. You cannot remove a custom user profile property from a user that is not authenticated. You must invoke `Sitecore.Security.UserProfile.Save()` after calling this method. For example, to remove the custom property named `attributeKey` from the profile of the context user:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
```

```
        Sitecore.Security.UserProfile profile = user.Profile;
        profile.RemoveCustomProperty("attributeKey");
        profile.Save();
```

3.4 How to Extend the Default User Profile

You can extend the default user profile to add custom user profile properties. If you extend the default user profile, you can access custom user profile properties through the User Manager in addition to the APIs described in the previous section How to Access Custom User Profile Properties.

To extend the default user profile:

1. In the Sitecore Desktop, select the Core database.[16]
2. In the Template Manager or the Content Editor, navigate to edit the `/Sitecore/Templates/System/Security/User` data template definition item.
3. Add any sections and fields and save changes to the new data template.
4. In the Sitecore Desktop, select the Master database.

In the User Manager, double-click a user, and then click the Profile tab to access extended profile properties.

[16] For instructions to select a database in the Sitecore Desktop, see the Client Configuration Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206.aspx.

3.5 Implement a Custom User Profile

This section contains procedures to implement a custom user profile.

3.5.1 How to Create a Custom User Profile

To create a custom user profile:

1. In the Sitecore Desktop, select the Core database.[17]
2. In the Template Manager or the Content Editor, duplicate the `/Sitecore/Templates/System/Security/User` data template definition item.
3. Add any fields and save changes to the new data template.[18]
4. In the Content Editor, select `/Sitecore/System/Settings/Security/Profiles`.
5. Insert a new user profile definition item using the custom user profile data template. To set this user profile as the default for new users created through the User Manager, sort the custom user profile definition item first.
6. In the Sitecore Desktop, select the Master database.

Important
When you create a new user using the User Manager, select the appropriate user profile in the User Profile field.

3.5.2 How to Apply a Custom User Profile Using the User Manager

To apply a custom user profile to a user using the User Manager:

1. In the User Manager, select the user.
2. In the Users group, click the Edit command. The Edit User dialog appears.
3. In the Edit User dialog, click the Profile tab.
4. Click Change. The Change User Profile dialog appears.
5. In the Change User Profile dialog, select the custom user profile, and then click Change.

3.5.3 How to Apply a Custom User Profile Using APIs

The `Sitecore.Security.UserProfile.ProfileItemId` property contains the ID of a user profile definition item in the core database. You must invoke `Sitecore.Security.UserProfile.Save()` after setting this property. For example, to set the custom profile definition item for the context user to the custom user profile definition item `/sitecore/system/settings/security/profiles/customuserprofile` in the core database:

```
using (new SecurityDisabler())
{
 string profilePath = "/sitecore/system/settings/security/profiles/customuserprofile";
 Sitecore.Security.Accounts.User user = Sitecore.Context.User;
 Sitecore.Data.Database dbCore = Sitecore.Configuration.Factory.GetDatabase("core");
 Sitecore.Data.Items.Item profileItem = dbCore.GetItem(profilePath);
 user.Profile.ProfileItemId = profileItem.ID.ToString();
 user.Profile.Save();
```

[17] For instructions to select a database in the Sitecore Desktop, see the Client Configuration Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206.aspx.
[18] For instructions to add fields to a data template, see the Data Definition Cookbook at http://sdn.sitecore.net/Reference/Sitecore%206.aspx.

```
        }
```

Note
The `Sitecore.Security.UserProfile.ProfileItemId` property contains the ID of an item based on a data template, not the ID of the data template itself.

Tip
You can specify the default profile item ID using the `defaultProfileItemId` attribute of each `/domains/domain` element in `/App_Config/Security/Domains.Config`.

3.5.4 How to Implement a Custom User Profile Class

To implement a custom user profile class to replace the default `Sitecore.Security.UserProfile` class exposed by the `Sitecore.Security.Accounts.User.Profile` property:

1. Create a custom user profile class that inherits from `Sitecore.Security.UserProfile`. You can use the following code sample:

```
namespace Namespace.Security
{
  public class UserProfile : Sitecore.Security.UserProfile
  {
    public string PropertyName
    {
      get
      {
        return GetCustomProperty("propertyname");
      }
      set
      {
        SetCustomProperty("propertyname", value);
        Save();
      }
    }
  }
}
```

2. Update the `inherits` attribute of the `/configuration/system.web/profile` element in `web.config` to the signature of the custom user profile class:

```
<profile defaultProvider="sql" enabled="true"
  inherits="Namespace.Security.UserProfile,Assembly">
```

3. Access the `Sitecore.Security.Accounts.User.Profile` property using the custom user profile class:

```
Namespace.Security.UserProfile profile = Sitecore.Context.User.Profile
  as Namespace.Security.UserProfile;

if(profile!=null)
{
  //TODO: handle profile.PropertyName
}
```

3.6 Sample User Profile Management Form

Many Web sites maintain profiles containing various data about users. You can implement a sublayout to contain a profile management form with code-behind to maintain user profiles, including changing passwords.

Note
You can also use the ASP.NET ChangePassword Web control to allow users to change their passwords.[19] For more information about the ASP.NET ChangePassword Web control, see the following section How to Use the ASP.NET ChangePassword Web Control.

Note
Do not grant unauthenticated access to the user profile management form or any page allowing the user to change their profile or password.

The following sample code for a sublayout file implements a very simple profile management data entry form:

```
<%@ Control Language="C#" AutoEventWireup="true" CodeBehind="Profile.ascx.cs"
  Inherits="Namespace.Web.UI.Profile" %>

Comment: <asp:textbox id="txtComment" runat="server" /><br />
Password: <asp:textbox id="txtPassword" runat="server" textmode="password"/><br />
New Password: <asp:textbox id="txtNewPassword" runat="server" textmode="password"/><br
/>
Confirm New Password:<asp:textbox id="txtNewPasswordConfirm" runat="server"
  textmode="password"/><br />
<asp:button id="btnGo" text="Go" runat="server" /><br />
<asp:label id="lblMessage" runat="server" />
```

This profile management form contains:

- A text field for the user to enter a comment to store in their profile.
- A password field for the user to enter their existing password.
- A password field for the user to enter a new password.
- A password field for the user to confirm the new password, to help ensure they did not enter their password erroneously.
- A button to submit the form.
- A label to contain any error message that results from submitting the form.

The following sample code for a sublayout code-behind file implements logic to update the user's profile:

```
using System;

namespace Namespace.Web.UI
{
  public partial class Profile : System.Web.UI.UserControl
  {
    protected void Page_Load(object sender, EventArgs e)
    {
      if(!Sitecore.Context.IsLoggedIn)
      {
        Sitecore.Web.WebUtil.Redirect("/");
      }
      else
      {
        if(IsPostBack)
        {
```

[19] For more information about the ASP.NET ChangePassword Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.changepassword.aspx.

```
                lblMessage.Text = "No change to implement.";

                if(String.IsNullOrEmpty(txtPassword.Text))
                {
                  lblMessage.Text = "Existing password required.";
                }
                else if(txtNewPassword.Text != txtNewPasswordConfirm.Text)
                {
                  lblMessage.Text = "Passwords do not match.";
                }
                else
                {
                  Sitecore.Security.Authentication.AuthenticationHelper authHelper =
                    new Sitecore.Security.Authentication.AuthenticationHelper(
                    Sitecore.Security.Authentication.AuthenticationManager.Provider);

                  try
                  {
                    if(!authHelper.ValidateUser(Sitecore.Context.User.Name,
txtPassword.Text))
                    {
                      throw new System.Security.Authentication.AuthenticationException(
                        "Incorrect password.");
                    }
                    else
                    {
                      if(txtComment.Text != Sitecore.Context.User.Profile.Comment)
                      {
                        Sitecore.Context.User.Profile.Comment = txtComment.Text;
                        Sitecore.Context.User.Profile.Save();
                        lblMessage.Text = "Comment changed.";
                      }

                      if((!String.IsNullOrEmpty(txtNewPassword.Text))
                        || !String.IsNullOrEmpty(txtNewPasswordConfirm.Text))
                      {
                        System.Web.Security.MembershipUser user
                          = System.Web.Security.Membership.GetUser(
                          Sitecore.Context.User.Name);

                        if(user.ChangePassword(txtPassword.Text, txtNewPassword.Text))
                        {
                          lblMessage.Text = "Password changed.";
                        }
                        else
                        {
                          throw new System.Security.Authentication.AuthenticationException(
                            "Unable to change password");
                        }
                      }
                    }
                  }
                  catch(System.Security.Authentication.AuthenticationException)
                  {
                    lblMessage.Text = "Processing error.";
                  }
                }
              }
              else
              {
                txtComment.Text = Sitecore.Context.User.Profile.Comment;
              }
            }
          }
        }
      }
```

The logic behind this code is as follows:

1. If the user has not authenticated, then redirect to the home page, and do nothing else. The user must authenticate before updating their profile.

2. If the page is not posting back, then the user has not had a chance to enter data into the form. In this case, populate the comment field with the comment from the user's profile, and do nothing else.
3. Update the error message to a default value indicating that no profile attributes have changed.
4. If the user has not entered a password, then display an error message and do nothing else.
5. If the user has entered a new password in either the new password field or the confirm new password field, and the values of those fields do not match, then display an error message, and do nothing else.
6. If the password entered by the user is not valid for the context user, then throw an exception, which will display a generic error message, and do nothing else.
7. If the comment entered by the user differs from the content stored in the user's profile, then update the comment in the user's profile, and display a message indicating that the comment has changed.
8. If the user has entered a new password, and the system is able to update the password associated with the user, then display a message indicating that the password has changed, and do nothing else.
9. If the system was not able to change the password associated with the user, then throw an exception, which will display a generic error message, and do nothing else.

Note
If you require authentication to access the profile management page, you do not need to require a password when the user updates their profile, especially when a user self-registers. For security, you should require a password if the user does not submit the profile change within a reasonable period after accessing the form. Require a password to prevent others from updating a user's profile if that user fails to log off before another user accesses the browser. Requiring the password is especially important when updating the user's password.

3.6.1 How to Use the ASP.NET ChangePassword Web Control

You can allow users to change their passwords using the ASP.NET ChangePassword Web control instead of writing code-behind.[20]

For example, add an ASP.NET ChangePassword Web control to a sublayout:

```
<asp:changepassword id="changePasswordControl" runat="server" />
```

The ASP.NET Login Web control changes the password for the context user without any custom code-behind.

[20] For more information about the ASP.NET ChangePassword Web control, see http://msdn.microsoft.com/en-us/library/system.web.ui.webcontrols.changepassword.aspx.

Chapter 4

Access Rights Management

This chapter describes solutions and provides sample code to manage access rights. It provides an overview of access rights, two techniques to defeat access rights, and example code to update access rights for an item.

This chapter contains the following sections:

- Overview of Access Rights
- User Switcher
- Security Disabler
- Apply Access Rights

4.1 Overview of Access Rights

Access rights control which users and roles can perform various operations on data in items, including reading and writing of field values and insertion of child items. If code accesses an item to which the context user does not have read access, the system behaves as if that item does not exist. If code attempts to update an item to which the context user has read but not write access, the system throws an exception. If code attempts to create an item under an item to which the context user has read, but not insert access, the system throws an exception.

In some cases, you may wish to allow a block of code to perform a certain operation, despite the fact that the context user does not have access rights to accomplish that task. In this case, you can use a user switcher to cause a segment of code to run in the context of a specific user as described in the following section User Switcher. Alternatively, you can use a security disabler to cause a segment of code to run in the context of a user with administrative rights as described in the following section Security Disabler.

You may also wish to update access rights associated with items and potentially their descendants as described in the following section Apply Access Rights.

4.2 User Switcher

You can use the `Sitecore.Security.Accounts.UserSwitcher` class to cause a segment of code to run in the context of a specific user, regardless of the context user. To use this approach, pass a `Sitecore.Security.Accounts.UserSwitcher` as a resource to a C# `using` statement.[21]

The `Sitecore.Security.Accounts.UserSwitcher` constructor sets the context user to the specified user. The code within the `using` statement block has the effective rights of the user specified by the first parameter passed to constructor of the the `Sitecore.Security.Accounts.UserSwitcher` class. By assigning roles to the user and applying access rights to items, you can control exactly which operations the block of code within the user switcher can perform on specific items. When the block ends, the `using` statement causes the .NET runtime engine to invoke the `Sitecore.Security.Accounts.UserSwitcher.Dispose()`, which resets the context user to the original context user.

For example, to invoke a segment of code as the user `user` in the domain `domain`:

```
string domainUser = @"domain\user";

if(Sitecore.Security.Accounts.User.Exists(domainUser))
{
  Sitecore.Security.Accounts.User user =
    Sitecore.Security.Accounts.User.FromName(domainUser,false);

  using(new Sitecore.Security.Accounts.UserSwitcher(user))
  {
    //TODO: code to invoke as user
  }
}
```

Note
To update items, you must first place the item in an editing state using `Sitecore.Data.Items.Item.Editing.BeginEdit()`. Afterwards, commit or rollback the transaction using `Sitecore.Data.Items.Item.Editing.EndEdit()` or `Sitecore.Data.Items.Item.Editing.CancelEdit()`. For example, to update the context item:

```
string domainUser = @"domain\user";

if(Sitecore.Security.Accounts.User.Exists(domainUser))
{
  Sitecore.Security.Accounts.User user =
    Sitecore.Security.Accounts.User.FromName(domainUser,true);

  using(new Sitecore.Security.Accounts.UserSwitcher(user))
  {
    Sitecore.Data.Items.Item contextItem = Sitecore.Context.Item;
    contextItem.Editing.BeginEdit();

    try
    {
      //TODO: update contextItem
      contextItem.Editing.EndEdit();
    }
    catch(Exception ex)
    {
      contextItem.Editing.CancelEdit();
    }
  }
}
```

[21] For more information about the C# `using` statement, see http://msdn.microsoft.com/en-us/library/yh598w02.aspx.

4.3 Security Disabler

You can use the `Sitecore.SecurityModel.SecurityDisabler` class to cause a segment of code to run in the context of a user with administrative rights, regardless of the context user. To use this approach, pass a `Sitecore.SecurityModel.SecurityDisabler` as a resource to a C# `using` statement. The code within the `using` statement block has full control of the entire system, and can take any action on any item or field.

For example, to invoke a segment of code in a security context with administrative rights:

```
using(new Sitecore.SecurityModel.SecurityDisabler())
{
  //TODO: code to invoke as administrator
}
```

Note
To update items, you must first place the item in an editing state using `Sitecore.Data.Items.Item.Editing.BeginEdit()`. Afterwards, commit or rollback the transaction using `Sitecore.Data.Items.Item.Editing.EndEdit()` or `Sitecore.Data.Items.Item.Editing.CancelEdit()`. For example, to update the context item:

```
using(new Sitecore.SecurityModel.SecurityDisabler())
{
  Sitecore.Data.Items.Item contextItem = Sitecore.Context.Item;
  contextItem.Editing.BeginEdit();

  try
  {
    //TODO: update contextItem
    contextItem.Editing.EndEdit();
  }
  catch(Exception ex)
  {
    contextItem.Editing.CancelEdit();
  }
}
```

4.4 Apply Access Rights

You can use code such as the following to set access rights for items in a Sitecore database.

```
private void SetRight(Sitecore.Data.Items.Item item,
  Sitecore.Security.Accounts.Account account,
  Sitecore.Security.AccessControl.AccessRight right,
  Sitecore.Security.AccessControl.AccessPermission rightState,
  Sitecore.Security.AccessControl.PropagationType propagationType)
{
  Sitecore.Security.AccessControl.AccessRuleCollection accessRules =
    item.Security.GetAccessRules();

  if(propagationType == Sitecore.Security.AccessControl.PropagationType.Any)
  {
    accessRules.Helper.RemoveExactMatches(account, right);
  }
  else
  {
    accessRules.Helper.RemoveExactMatches(account, right, propagationType);
  }

  if(rightState != Sitecore.Security.AccessControl.AccessPermission.NotSet)
  {
    if(propagationType == Sitecore.Security.AccessControl.PropagationType.Any)
    {
      accessRules.Helper.AddAccessPermission(account, right,
        Sitecore.Security.AccessControl.PropagationType.Entity, rightState);
      accessRules.Helper.AddAccessPermission(account, right,
        Sitecore.Security.AccessControl.PropagationType.Descendants, rightState);
    }
    else
    {
      accessRules.Helper.AddAccessPermission(account, right, propagationType,
        rightState);
    }
  }

  item.Security.SetAccessRules(accessRules);
}

private void SetRight(string strDatabase, string strItem, string strAccount,
  string strRight, Sitecore.Security.AccessControl.AccessPermission rightState,
  Sitecore.Security.AccessControl.PropagationType propagationType)
{
  Sitecore.Data.Database db = Sitecore.Configuration.Factory.GetDatabase(strDatabase);
  Sitecore.Data.Items.Item item = db.GetItem(strItem);
  Sitecore.Security.Accounts.AccountType accountType =
    Sitecore.Security.Accounts.AccountType.User;

  if(Sitecore.Security.SecurityUtility.IsRole(strAccount))
  {
    accountType = Sitecore.Security.Accounts.AccountType.Role;
  }

  Sitecore.Security.Accounts.Account account =
    Sitecore.Security.Accounts.Account.FromName(strAccount, accountType);
  Sitecore.Security.AccessControl.AccessRight right =
    Sitecore.Security.AccessControl.AccessRight.FromName(strRight);
  SetRight(item, account, right, rightState, propagationType);
}
```

This sample code includes two methods. The first method accepts object parameters. The second method accepts string parameters, converts them to objects, and then calls the first method. The logic behind the first method is as follows:

1. Retrieve access rules for the item.
2. If the caller specifies `PropagationType` of `Any`, the access rule changes apply to the item and its descendants. In this case, the code removes access rights for the specified account from the specified item and its descendants. Otherwise, the code removes access rights for

the specified account from the specified item, and possibly its descendants depending on the specified propagation type.

3. If the caller specifies `AccessPermission` of `NotSet`, the code does not apply new access rights for the account. Otherwise, if the caller specifies `PropagationType` of `Any`, the code applies the specified access right for the specified account to the item and its descendants. If the caller specifies any other `PropagationType`, the code applies access rights based on that value.
4. Commit access right changes to the item.

Important
This code does not account for roles. Removing access rights for a user does not remove access for any of its roles, and removing access for a role does not affect nested roles.

Chapter 5

System.Web.Security APIs

This chapter describes .NET APIs to implement common security operations that are not exposed by Sitecore APIs.

This chapter contains the following sections:

- System.Web.Security.Roles
- System.Web.Security.MembershipUser
- System.Web.Security.Membership

5.1 System.Web.Security.Roles

The `System.Web.Security.Roles` class exposes the security features that are not abstracted by the Sitecore security APIs as described in the following sections.

5.1.1 System.Web.Security.Roles.CreateRole()

The `System.Web.Security.Roles.CreateRole()` method creates a role in a domain. For example, to create the role `role` in the domain `domain` if that role does not already exist:

```
string domainRole = @"domain\role";

if(!Sitecore.Security.Accounts.Role.Exists(domainRole))
{
  System.Web.Security.Roles.CreateRole(domainRole);
}
```

5.1.2 System.Web.Security.Roles.DeleteRole()

The `System.Web.Security.Roles.DeleteRole()` method removes all members from the role specified by the first parameter, and then removes that role. For example, to remove the role `role` from the domain `domain`:

```
string domainRole = @"domain\role";

if(Sitecore.Security.Accounts.Role.Exists(domainRole))
{
  System.Web.Security.Roles.DeleteRole(domainRole);
}
```

Note
Depending on the number of users, deleting a role can be a long-running operation.

5.2 System.Web.Security.MembershipUser

The underlying .NET membership provider uses the `System.Web.Security.MembershipUser` class to represent users. The following sections describe functions available in this class that the Sitecore security APIs do not expose.

5.2.1 System.Web.Security.MembershipUser.GetUser()

The `System.Web.Security.MembershipUser.GetUser()` method returns the `System.Web.Security.MembershipUser` specified by the first parameter. For example, to access the context user as a `System.Web.Security.MembershipUser`:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
System.Web.Security.MembershipUser mUser =
  System.Web.Security.Membership.GetUser(user.Name);
```

5.2.2 System.Web.Security.MembershipUser.ChangePassword()

The `System.Web.Security.MembershipUser.ChangePassword()` method changes the password for the user, or returns False if it is unable to change the password for the user. For example, to set the password for the context user to `newPassword` when the old password is `oldPassword`:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
System.Web.Security.MembershipUser mUser =
  System.Web.Security.Membership.GetUser(user.Name);

if (!mUser.ChangePassword("oldPassword","newPassword"))
{
  //TODO: handle case that password was not changed
}
```

5.2.3 System.Web.Security.MembershipUser.ChangePasswordQuestionAndAnswer()

The `System.Web.Security.MembershipUser.ChangePasswordQuestionAndAnswer()` method changes the password, security question, and answer for the user, or returns False if it the system is unable to make these changes. For example, to set the password for the context user to `newPassword`, their security question to `newQuestion`, and their answer to that question to `newAnswer`:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
System.Web.Security.MembershipUser mUser =
  System.Web.Security.Membership.GetUser(user.Name);

if (!mUser.ChangePasswordQuestionAndAnswer("newPassword","newQuestion","newAnswer"))
{
  //TODO: handle case that password was not changed
}
```

5.2.4 System.Web.Security.MembershipUser.ResetPassword()

The `System.Web.Security.MembershipUser.ResetPassword()` method changes the password for the user to a random string, and returns that string. For example, to randomize the password for the context user:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
System.Web.Security.MembershipUser mUser =
  System.Web.Security.Membership.GetUser(user.Name);
string password = mUser.ResetPassword();
```

The `System.Web.Security.MembershipUser.ResetPassword()` method provides an additional signature for systems that require the answer to the user's security password. For example, to randomize the password for the context user, when then answer to their security question is `answer`:

```
Sitecore.Security.Accounts.User user = Sitecore.Context.User;
System.Web.Security.MembershipUser mUser =
  System.Web.Security.Membership.GetUser(user.Name);
string password = mUser.ResetPassword("answer");
```

5.2.5 System.Web.Security.MembershipUser.UnlockUser()

The `System.Web.Security.MembershipUser.UnlockUser()` method unlocks a user locked out due to entering an invalid password beyond the number of times allowed. If the system is not able to unlock the user, this method returns False; otherwise it returns true. For example, to unlock the user `user` in the domain `domain`:

```
string domainUser = @"domain\user";

if(Sitecore.Security.Accounts.User.Exists(domainUser))
{
  System.Web.Security.MembershipUser mUser =
    System.Web.Security.Membership.GetUser(domainUser);

  if(!mUser.UnlockUser())
  {
    //TODO: handle case that system is not able to unlock user
  }
}
```

For information about configuring the number of times a user may enter an invalid password before becoming locked out, see the section Membership Provider Configuration.

5.3 System.Web.Security.Membership

The following sections describe functions available in the `System.Web.Security.Membership` class that the Sitecore security APIs do not expose.

5.3.1 System.Web.Security.Membership.GetUserNameByEmail()

Assuming there is only one such user, the `System.Web.Security.Membership.GetUserNameByEmail()` method retrieves the name of the user associated with the email address specified by the first parameter. For example, to process the `Sitecore.Security.Acconts.User` associated with the email address `address@domain.tld`

```
string domainUser =
  System.Web.Security.Membership.GetUserNameByEmail("address@domain.tld");

if((!String.IsNullOrEmpty(domainUser))
  && Sitecore.Security.Accounts.User.Exists(domainUser))
{
  Sitecore.Security.Accounts.User user =
    Sitecore.Security.Accounts.User.FromName(domainUser,false);
  //TODO: handle user
}
```

5.3.2 System.Web.Security.Membership.FindUsersByEmail()

The `System.Web.Security.Membership.FindUsersByEmail()` method returns a list of the `System.Web.Security.MembershipUser` objects associated with the email address specified by the first parameter. For example, to process the `Sitecore.Security.Accounts.User` associated with the email address `address@domain.tld`:

```
foreach(System.Web.Security.MembershipUser mUser in
  System.Web.Security.Membership.FindUsersByEmail("address@domain.tld"))
{
  Sitecore.Security.Accounts.User user =
    Sitecore.Security.Accounts.User.FromName(mUser.UserName,false);
  //TODO: handle user
}
```

Chapter 6

Appendix A

This appendix documents how symbols map to elements in the API.

This appendix contains the following section:

- Sitecore.Security.AccessControl.AccessRight

6.1 Sitecore.Security.AccessControl.AccessRight

The `Sitecore.Security.AccessControl.AccessRight` class represents an individual access right as defined under the `/configuration/sitecore/accessRights` element in `web.config`. The following properties of the `Sitecore.Security.AccessControl.AccessRight` class represent the various access rights.

Property	Access Right Code
Any	*
FieldRead	field:read
FieldWrite	field:write
InsertShow	insert:show
ItemAdmin	item:admin
ItemCreate	item:create
ItemDelete	item:delete
ItemRead	item:read
ItemRename	item:rename
ItemWrite	item:write
LanguageRead	language:read
LanguageWrite	language:write
SiteEnter	site:enter
WorkflowCommandExecute	workflowCommand:execute
WorkflowStateDelete	workflowState:delete
WorkflowStateWrite	workflowState:write