Microsoft® Official Course

Module 2

Common SharePoint Artifacts & Tasks

Module Overview

- Developing Web Parts
- Using Event Receivers
- Using Timer Jobs
- Storing Configuration Data

Lesson 1: Developing Web Parts

- Introduction to Web Parts
- Understanding the Web Part Lifecycle
- Visual Web Parts
- Demonstration: Creating a Visual Web Part
- The .webpart File

Introduction to Web Parts

- Do not require app infrastructure or any special services/service applications to run
- Can expose and consume data with other web parts
- Inherit from the WebPart class
- Deployed as assembly, with a .webpart definition file
- Use app part instead where possible

Understanding the Web Part Lifecycle

- OnInit
- OnLoad
- CreateChildControls
- EnsureChildControls
- SaveViewState
- OnPreRender
- Page.PreRenderComplete
- Render
- RenderContents
- OnUnload

Visual Web Parts

- Inherit from the WebPart class
- Define interface in a custom control (ascx) file
- Expose an additional lifecycle method *Page_Load*
- Available in both farm and sandboxed solutions

Demonstration: Creating a Visual Web Part

- In this demonstration, you will see how to:
  - Add a visual web part to a Visual Studio solution
  - Add a control to a visual web part
  - Edit control properties
  - Debug a visual web part by using Visual Studio

The .webpart File

- XML file containing a web part definition
- Stored in web part gallery
- Identifies the assembly and type which contains the web part
- Contains web part metadata such as the web part name, description, and display group

WebPart Definition File

```
<webParts>
    <webPart xmlns="http://schemas.microsoft.om/WebPart/v3">
        <metaData>
            <!-- Add the fully qualifies type name and strong named assembly name -->
            <type name="ContosoWebPartNamespace.ContosoWebPart, WebPartAssembly Strong
Name>
            <!-- Add an error message to display if the web part does not import
correctly -->
            <importErrorMessage>Error importing web part.</importErrorMessage>
        </metadata>
        <data>
            <properties>
                <!-- Add properties which define the web part metadata -->
                <property name="Title" type="string">Contoso Web Part</property>
                <property name="Description" type="string">This is the Contoso Web
Part</property>
            </properties>
        </data>
    </webPart>
</webParts>
```

Lesson 2: Using Event Receivers

- Introduction to Event Receivers
- Developing an Event Receiver
- Deploying an Event Receiver
- Discussion Question

Introduction to Event Receivers

- Handle External Actions
- Event Hosts
  - SPSite
  - SPWeb
  - SPList
  - SPContentType
- Event Synchronization
- Before Events (ing)
- After Events (ed)

Developing an Event Receiver

- Inherit from Base Class
  - SPWebEventReceiver
  - SPListEventReceiver
  - SPItemEventReceiver
  - SPEmailEventReceiver
  - SPWorkflowEventReceiver
  - SPSecurityEventReceiver
- Override Methods
- Cancel Before Events
  - Without an error message
  - With an error message
  - With a page redirect

Deploying an Event Receiver

- Deploy manually (bad practice)
- Deploy with a solution and Feature (best practice)
- Bind event receiver to an event host with code
- Bind event receiver to an event host declaratively
- Common properties
  - Assembly
  - Class
  - Type
  - Name (optional)
  - Synchronization (optional)
  - SequenceNumber (optional)

Bind Event Receiver using Code

```
using (SPSite site = new SPSite("http://portal.contoso.com"))
{
    using (SPWeb web = site.OpenWeb())
    {
        // Obtain a reference to the event host.
        SPList list = web.Lists["Suppliers"];
        // Create a new instance of the SPEventReceiverDefinition class by using the Add
method of the
        // EventReceivers collection of the event host.
        SPEventReceiverDefinition eventRec = list.EventReceivers.Add();

        // Optionally, specify a name for the event receiver binding.
        eventRec.Name = "Event Receiver For ItemAdded";
        // Specify the  strong name for the assembly which contains the complied event
receiver.
        eventRec.Assembly =
          "ReceiverAssembly, Version=1.0.0.0, Culture=Neutral,
PublicKeyToken=24a5ef6a3fe2c28c";
        // Specify the fully qualified type name for your event receiver class.
        eventRec.Class = "ReceiverNamespace.ReceiverClass";
        // Specify the event which should be handled.
        eventRec.Type = SPEventReceiverType.ItemAdded;
        // Specify whether the event should run synchronously or asynchronously.
        eventRec.Synchronization = SPEventReceiverSynchronization.Synchronous;
        // Optionally, specify a sequence value for the event receiver.
        eventRec.SequenceNumber = 50;
        // Call the Update method to persist the changes.
        eventRec.Update();
    }
}
```

Lesson 3: Using Timer Jobs

- Introduction to Timer Jobs
- Developing a Timer Job
- Deploying a Timer Job
- Demonstration: Examining Timer Job Schedules
- Work Item Timer Jobs
- Developing a Work Item Timer Job

Introduction to Timer Jobs

- Long running processes
- Processor intensive processes
- One time or regular schedule
- Reduce instantaneous load on server farm
- OWSTIMER.exe

Developing a Timer Job

- Create a class which derives from SPJobDefinition
- Add a default constructor
- Add a non-default constructor
  - Name (string)
  - Web application (SPWebApplication)
  - Server (SPServer)
  - Lock Type (SPJobLockType)
- Override the Execute method

Timer Job Pattern

```
// Define a class which inherits from the SPJobDefinition class.
public class ContosoTimerJob : SPJobDefinition
{
    // Define a default constructor.
    public ContosoTimerJob()
    {
    }
    // Define a non-default constructor
public ContosoTimerJob(string name, SPWebApplication webApplication,
        SPServer server, SPJobLockType lockType)
        : base(name, webApplication, server, lockType) // Call the constructor of the
base class.
    {
    }
    public override void Execute(Guid targetInstanceId)
    {
        // Add custom logic here.
    }
}
```

Deploying a Timer Job

- Use a Feature and Feature receiver
- Create a new instance of the job
- Create a schedule
  - SPMinuteSchedule
  - SPHourlySchedule
  - SPDailySchedule
  - SPWeeklySchedule
  - SPMonthlySchedule
  - SPYearlySchedule
- Assign the schedule to the job
- Call the Update method

Timer Job Deployment

```
public class TimerJobInstaller : SPFeatureReceiver
{
    // Define a constant with the job name.
    const string JobName = "ContosoTimerJob";
    // Create a method to remove the job if it is installed.
    private void RemoveJob(SPWebApplication webApplication)
    {
        foreach (var job in webApplication.JobDefinitions)
        {
            if (job.Name.Equals(JobName))
            {
                job.Delete();
            }
        }
    }
    // Override the FeatureActivated method to install the job.
    public override void FeatureActivated(SPFeatureReceiverProperties properties)
    {
        // Obtain a reference to the web application.
        SPWebApplication webApplication =
((SPSite)properties.Feature.Parent).WebApplication;
        // Remove the timer job if it is already installed.
        RemoveJob(webApplication);
        // Create a new instance of the timer job.
        ContosoTimerJob timerJob = new ContosoTimerJob(JobName, webApplication);
        // Create a schedule for the job.
        SPMinuteSchedule schedule = new SPMinuteSchedule();
        schedule.BeginSecond = 0;
        schedule.EndSecond = 5;
        schedule.Interval = 10;
        // Assign the schedule to the job.
        timerJob.Schedule = schedule;
        // Call the Update method to persist the changes.
        timerJob.Update();
    }
    // Override the FeatureDeactivated method to uninstall the job.
    public override void FeatureDeactivated(SPFeatureReceiverProperties properties)
    {
        // Obtain a reference to the web application.
        SPWebApplication webApplication =
((SPSite)properties.Feature.Parent).WebApplication;
        // Remove the timer job.
        RemoveJob(webApplication);
    }
}
```

Demonstration: Examining Timer Job Schedules

- In this demonstration, you will review:
  - The timer jobs included in SharePoint 2013
  - The timer service process

Work Item Timer Jobs

- Process a set (or queue) of items
- Queue managed by SharePoint
- Use code to add items to the queue
- Timer job run on a schedule
- Batch or single item processing

Developing a Work Item Timer Job

- Create a class which derives from SPWorkItemJobDefinition
- Add a default constructor
- Add a non-default constructor
  - Name (string)
  - Web application (SPWebApplication)
- Override the WorkItemType method
- Override the DisplayName property
- Optionally override the BatchFetchLimit property
- Override either the ProcessWorkItem or the ProcessWorkItems method

Work Item Timer Job

```
public class WorkItemTimerJob : SPWorkItemJobDefinition
{
    public WorkItemTimerJob()
    {
    }
    public WorkItemTimerJob(string name, SPWebApplication webApplication
        : base(name, webApplication)
    {
    }
    public override Guid WorkItemType()
    {
        return new Guid("{7B1F0F56-2A2D-4AA3-B962-FCAD66ED02C8}");
    }
    public override string DisplayName
    {
        get
        {
            return "Contoso Work Item Timer Job";
        }
    }
    public override int BatchFetchLimit
    {
        get
        {
            return 100;
        }
    }
    public override bool ProcessWorkItem(SPContentDatabase contentDatabase,
                                                                  SPWorkItemCollection
workItems,
                                                                  SPWorkItem workItem,
                                                                  SPJobState jobState)

    {
        // Add code to process work item.
        // Delete the work item.
        workItems.DeleteWorkItem(workItem.Id);
    }
}
```

Lesson 4: Storing Configuration Data

- Configuration Storage Options
- Using Property Bags
- Manipulating Web.config Files
- Storing Hierarchical Data
- Discussion Question

Configuration Storage Options

- Configuration data requirements
- Many Options
  - SharePoint list
  - Property bags
  - Configuration files
  - Persisted objects
- Considerations
  - How often configuration changes
  - Scope of configuration data
  - Visibility of data to users

Using Property Bags

```
using (SPWeb web  = SPContext.Current.Site.RootWeb)
{
    // Always check an item does not already exist before attempting to add an item.
    if(web.Properties["Key"] == null)
    {
        // Add a new item to the property bag.
        web.Properties.Add("Key", "Value");
        web.Properties.Update();
    }
    // Retrieve a value stored in the property bag.
    string storedValue = web.Properties["Key"];
    // Update an item store in a property bag.
    web.Properties["Key"] = "NewValue";
    web.Properties.Update();
    // Remove an item from the property bag.

    web.Properties["Key"] = null;
    web.Properties.Update();
}
```

Manipulating Web.config Files

- Data with low change frequency
- Requires IIS restart
- Not compatible with timer jobs or sandboxed solutions
- Use SPWebConfigModification class to deploy changes to all servers

web.config

```
// Create a new instance of the SPWebConfigModification class.
SPWebConfigModification modification = new SPWebConfigModification();
// Set properties of the SPWebConfigModification instance.
// For example, modification.Path and modification.Value
// Obtain a  reference to the web applicationfor which you need to change the
configuration file.
SPSite site = new SPSite("http://sharepoint.contoso.com);
SPWebApplication webApplication =site.WebApplication;
//Add the SPWebConfigModification instance to the list of modifications for the web
application.
webApplication.WebConfigModifications.Add(modification);
// Call the web applications Update method to save the changes.
webApplication.Update();
// Obtain a reference to the content service.
SPWebService service = SPWebService.ContentService;
// Call the ApplyWebConfigModifications method to apply the changes to the farm.
service.ApplyWebConfigModifications();
```

Storing Hierarchical Data

- Inherit from SPPersistedObject, and annotate with Guid attribute
- Include default and non-default constructors
  - public PersistedClass() { }
  - Public PersistedClass(string name, SPPersistedObject parent) : base (name, parent) { }
- Add Persisted attributes to fields (not properties)
- Use SPPersistedObject.GetChild<>() to retrieve values
- Require read and write permissions on the configuration database

Persisted Object

```
// Annotate the class with a Guid attribute.
[Guid("EAD5BAE5-3C66-42B4-B0CC-039AA67373A4")]
public class PersistedClass : SPPersistedObject  // Inherit from the SPPersistedObject
class.
{
    // Always include a default constructor.
    public PersistedClass()
    {
    }
    // Always include a non-default constructor.
    public PersistedClass(string name, SPPersistedObject parent)
        : base(name, parent) // Class the base class constructor.
    {
    }
    [Persisted] // Annotate fields which must be stored.
    string Value1;
    [Persisted] // Annotate fields which must be stored.
    string Value2;
    [Persisted] // Annotate fields which must be stored.
    int Value 3;
    // Add methods and properties.
    ...
}
```

Using Persisted Object

```
// Obtain a reference to the parent item.
SPFarm farm = SPFarm.Local;
// Create a root item.
// Create a new instance of the class by specifying the data item name, and root item.

 PersistedClass rootClass = new PersistedClass("ContosoRootItem", farm);
 // Set properties on the item.
 rootClass.Value1 = "Stored Data";
 rootClass.Value2 = "More Store Data";
 rootClass.Value3 = 4;
 // Call the Update method to persist the changes.
 rootClass.Update();
 // Create a new item as a child of the root item.
 // Create a new instance of the class by specifying the data item name, and root item.
 PersistedClass childClass = new PersistedClass("ContosoChildIte2m", rootClass);
 // Set properties on the item.
 childClass.Value1 = "Stored Data Item";
 // Call the Update method to persist the changes.
 childClass.Update();
 // Retrieve an item from the hierarchical data store.
 //Obtain a reference to the root item.
 SPFarm farm = SPFarm.Local;
 // Use the generic GetChild method to retrieve the item.
 PersistedClass retrievedValue = farm.GetChild<PersistedClass>("ContosoRootItem");
```

Managing Permissions in SharePoint 2013

- Permissions Classes
- Checking Permissions
- Assigning Permissions
- Managing Access to Resources

How SharePoint Represents Users

- SPUser
- SPGroup
- SPPrincipal

```
SPUser user =
   SPContext.Current.Web.CurrentUser;

SPUserCollection users =
   SPContext.Current.Web.AllUsers;
```

Permissions Classes

- SPSecurableObject represents a list, library, website or item
- SPRoleDefinition represents a permissions level
- SPRoleAssignment represents the assignment of a permission level to a security principal such as a user or group

Permissions

```
using(SPWeb hrWeb = SPContext.Current.Site.AllWebs["HumanResources"])
{
   //Get the permission levels defined for this website
   SPRoleDefinitionCollection roleDefinitions = hrWeb.RoleDefinitions;
   //Get the collection of role assignments
   SPRoleAssignmentCollection roleAssignments = hrWeb.RoleAssignments;
   //Create a new role assignment
SPRoleAssignment newRoleAssignment = new
      SPRoleAssignment("CONTOSO\\HRManager","HRManager@contoso.com","HR Manager","");
   //Get the permission level binding for the new role assignment
SPRoleDefinitionBindingCollection roleDefinitionBindings =
      newRoleAssignment.RoleDefinitionBindings;
   //Add the role definition to the bindings for the assignment
   roleDefinitionBindings.Add(roleDefinitions["SpecialAccess"]);
   //Add the new role assignment to the collection of assignments in the site
   roleAssignments.Add(newRoleAssignment);
}
```

Checking Permissions

```
//Get the current web site and user
using (SPWeb website = SPContext.Current.Web)
{
   SPUser user = website.CurrentUser;
   if (website.DoesUserHavePermissions(user.LoginName, SPBasePermissions.EditListItems))
   {
      //User can edit items in lists
      editButton.Visible = true;
   }
   else
   {
      //User cannot edit items in lists
      editButton.Visible = false;
   }
}
```

Assigning Permissions

- Assigning a Permissions Level
  - Create a new **SPRoleAssignment**
  - Add a role definition binding to the assignment
  - Add the assignment to the **RoleAssignments** collection on the securable object
- Creating a Custom Permissions Level
  - Create a new **SPRoleDefinition**
  - Add permissions to the **BasePermissions** collection
  - Add the role definition to the **RoleDefinitions** collection on the website

Assigning a Permissions Level

```
//Get the current web site
using (SPWeb website = SPContext.Current.Web)
{
   //Create the role assignment object
  SPRoleAssignment assignment = new SPRoleAssignment(website.SiteGroups["Viewers"]);
   //Add a binding to the Read permissions level
   assignment.RoleDefinitionBinding.Add(website.RoleDefinitions["Read"]);
   //Add the new role assignment to the website's assignments collection
   website.RoleAssignments.Add(assignment);
   //Update the site
   website.Update();
}
```

Creating a Custom Permissions Level

```
//Get the current web site
using (SPWeb website = SPContext.Current.Web)
{
   //Create the new permissions level
   SPRoleDefinition customPermissionsLevel = new SPRoleDefintion();
   //Set the name and base permissions for the permissions level
   customPermissionsLevel.Name = "EditButNotDelete";
   customPermissionsLevel.BasePermissions = SPBasePermissions.AddListItems |
      SPBasePermissions.OpenItems | SPBasePermissions.ViewListItems |
SPBasePermissions.EditListItems;
   //Add the permissions level to the website
   website.RoleDefinitions.Add(customPermissionsLevel);
   website.Update();
}
```

Managing Access to Resources

- Permissions Inheritance
  - Breaking inheritance
  - Restoring inheritance
- Anonymous Access
  - Enabling anonymous users to access a site
  - Assigning permissions to anonymous users