Robert L. Bogue
MS MVP, MCSE, MCSA:Security
317-844-5310
Rob.Bogue@ThorProjects.com

Using SharePoint Solutions to Deliver Server-Side Solutions

If you've ever worked on more than the most trivial installation program, you realize that developing installers isn't for the faint of heart. After spending days or weeks learning about the MSI file format, how to assemble tables, and how to add dialogs, you're left with a collection of documentation that reaches from the floor to the ceiling, and you can feel just as lost as when you began. Further, the experience doesn't help much when you need to deliver the software in a silent or automated way. You need to acquire additional knowledge to handle automated setup files.

At first glance you might wonder why there's a feature in SharePoint Technologies called "Solutions." The Solutions framework seems to be a direct competitor with the Microsoft Windows Installer technology—and its MSI file format—but it isn't. Where the Windows installer is targeted toward client-side application installation, the SharePoint Solutions approach is targeted toward delivering complete solutions to SharePoint Servers.

This discussion will walk you through the deployment of a semifictional application, *WFInspector*, which is a workflow inspector tool. The tool itself is of marginal value because it doesn't add much functionality beyond what is available out of the box; however, it's very representative of the kinds of solutions that you may want or need to deploy.

Features, Web Parts, and Site Definitions

It would be fair to ask why another "thing" is needed in SharePoint to deliver solutions. For instance, there's the now infamous SharePoint feature called "Features." Although Features is an immensely powerful technique for adding functionality to SharePoint, it isn't without its limitations. Features must be scoped at one level (farm, web application, site collection, or site), and sometimes creating a real solution requires more than one feature. In addition, it's sometimes necessary to deliver files for a feature outside of the Features' directory. If you need to deploy files into the *_layouts* directory tree, you've got files outside of the *Features* directory to track and deliver.

The setup just described is for delivering Features only. There are also two other items that people typically want to deliver, which aren't Features. Web Parts aren't deployed through Features, but they definitely have their own quirks concerning deployment, including code access security (CAS) policy, whether the assembly is delivered to the *bin* directory or to the global assembly cache (GAC), and the delivery of template Web Part files. Site definitions are a similarly complex item that require creating the site definition directory and adding the *WebTemp*.xml* file to the correct directory.

In short, there's no way to use Features to deliver complete functionality for a solution. There's more to it than that, and that's the reason why the Solutions mechanism was developed.

Why Create Solution Files?

The Solutions format is very simple. It contains one mandatory file, *manifest.xml*, packed into a CAB file format and renamed with the WSP extension. Perhaps the most difficult part of the whole process is creating a Diamond Directive File (DDF) that you can use to create the CAB file with the *MAKECAB.EXE* utility—and it's a variant of the old INI file format that we know and love. The net is that the barrier to entry is fairly low, and you get a way to install, and retract, complete solutions from the entire web farm in a way that allows the state of the web farm to be kept in sync by SharePoint.

In the end, a solution file is easier to manage than any other mechanism for deploying files to SharePoint servers. Once added to the solution store, it can be managed through a central administration interface, or through the command line. Solutions can be deployed and retracted at will.

Before walking through the creation of a sample solution file, it's important that you understand the three major components of what you're delivering:

- *WFInspector* feature – This feature uses the functionality to add pages to a web site. It also uses the functionality to add a link to these pages.
- *WFInspectorStaple* feature – This feature staples the *WFInspector* feature to every web site. In other words, the *WFInspectorStaple* makes sure that every new web site that is created will have the *WFInspector* functionality.
- *WFInspector* web part – This web part is the core of the solution. It provides a listing of the workflows running on the items in a given site.

With these components of the solution in mind, you can start building the *manifest.xml* file that the solution file needs.

Creating a Manifest File

The core file in the Solutions format is the *Manifest.xml*, an XML file that describes the contents of the WSP (solutions file). The root node of this XML file includes these attributes:

- *xmlns* – This attribute is the namespace for the node that, like most SharePoint XML files, is *http://schemas.microsoft.com/sharepoint/* for this node.
- *DeploymentServerType* – An attribute with the values *ApplicationServer* and *WebFrontEnd*; solutions that need to be deployed only on front-end web servers use the *WebFrontEnd* option, and solutions that need to be deployed on any of the application servers, such as the indexing server, use the *ApplicationServer* option.
- *ResetWebServer* – Set this attribute to *TRUE* typically, if you want the solution deployment framework to automatically reset the web server for you.
- *SolutionId* – This attribute specifies the GUID identifier for the solution, in registry format minus the braces.

Once the <Solution> node is in place, it's time to move on and add the features to the manifest file. Add them by using the <FeatureManifests> and <FeatureManifest> set of tags. The <FeatureManifests> tag is simply a container for multiple <FeatureManifest> tags, each of which includes a single attribute, the *Location* attribute, which refers to the *feature.xml* of the feature to be deployed. In this case there are two nodes that refer to the two solutions to be deployed. The <FeatureManifests> node for the solution example discussed here looks like this:
```
<FeatureManifests>
  <FeatureManifest Location="WFInspector\feature.xml" />
  <FeatureManifest Location="WFInspectorStaple\feature.xml" />
</FeatureManifests>
```

The SharePoint Solution framework will manage connecting the Element Manifests referred to in the *feature.xml* file for you. However, none of the ancillary files used by the feature, whether in the feature directory or not, will be copied by default; therefore, a different strategy for them is needed. Fortunately, the Solution framework provides two tags that can be used in *manifest.xml* to copy arbitrary files during deployment.

You can use a <RootFile> node in the <RootFiles> node to deploy files anywhere within *C:\Program Files\Common Files\Microsoft Shared\Web Server Extensions\12*. For brevity, this directory will be referred to as *…\12* hereafter in this discussion. You can use a <TemplateFile> node in the <TemplateFiles> node to deploy files within *…\12*. In this example you need to deploy a page, *default.aspx*, into the same directory as the Feature that deploys with a module element.

Because the file you need to deploy is under the *TEMPLATE* directory, use the <TemplateFile> node like this:

```
<TemplateFiles>
  <TemplateFile Location="FEATURES\WFInspector\default.aspx" />
</TemplateFiles>
```

Embedding CAS Policies

The final objective for the *manifest.xml* file is to deploy the web part, which is done in two pieces: deploying the assembly and deploying the *.webpart* template file that users employ to add the web part to a page. You can deploy the assembly with an <Assembly> node under the <Assemblies> node. The <Assembly> tag has two attributes:

- *DeploymentTarget* – This attribute specifies where the assembly (DLL) will be copied to. The *WebApplication* option makes a copy of the file in the *bin* directory of the web application. This mechanism means that the file will not be trusted fully and will be subject to CAS policies. The other value, *GlobalAssemblyCache*, does deploy the assembly to the GAC, which means it will be trusted fully and thus not subject to CAS policies. Note that the use

of the *GlobalAssemblyCache* attribute will require that the user doing the deployment explicitly allows GAC deployment.
- *Location* – As with the tags previously discussed, the *Location* attribute provides the location of the assembly file in the solution file.

Although it's not demonstrated here, it's possible to embed CAS policies in the solution file. During deployment the solutions framework will add these CAS policies to the existing policy file. The fact that the solution framework does this manipulation during deployment is a great advance over previous deployment scenarios. It doesn't require that administrators keep a master CAS policy or manually manipulate the XML files themselves.

In the case of web part assemblies, like the *WFInspectorWebParts* assembly that contains the web part for this example, you also need a <SafeControls> node containing one or more <SafeControl> nodes in the *manifest.xml* file and irrespective of whether a CAS policy is used or the assembly is deployed to the GAC. These <SafeControl> entries will be copied to the *web.config* of the application during the deployment process. Just like the <SafeControl> entry in *web.config*, the <SafeControl> element contains these four attributes:

- *Assembly* – This attribute is the fully specified assembly name that includes name, version, culture, and public key token.
- *Namespace* – This attribute is the namespace to which the web part or web parts belong.
- *TypeName* – This attribute is the name of the web part class—or more commonly, *, indicating that all classes in the namespace should be considered safe.
- *Safe* – This attribute must contain *TRUE* for the <SafeControl> entry to be considered.

In putting the web part assemblies together, you can deploy the XML fragment for deploying the assembly this way:

```
<Assemblies>
  <Assembly DeploymentTarget="WebApplication" Location=
    "WFInspectorWebParts.dll">
    <SafeControls>
      <SafeControl Assembly="WFInspectorWebParts, Version=1.0.0.0,
        Culture=neutral, PublicKeyToken=42c55593c85ca4ad"
        Namespace="WFInspectorWebParts" TypeName="*" Safe="True"/>
    </SafeControls>
  </Assembly>
</Assemblies>
```

Now it's time to direct SharePoint how to deploy the *.webpart* file, and then users can add the *WFInspectorWebParts* web part themselves, if they want to. They can add the web part with the <DwpFiles> and <DwpFile> set of tags. The <DwpFile> tag contains a single attribute: *Location*. It indicates the location of the file in the solution file. The "Dwp" part of the name is a holdover from previous versions of the product and was the extension used for web parts prior to SharePoint 3.0.

The whole *manifest.xml* file looks like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<Solution SolutionId="6C692C86-35FE-4967-8A6E-BACD9AB10423"
  DeploymentServerType="WebFrontEnd" ResetWebServer="TRUE"
  xmlns="http;//schemas.microsoft.com/sharepoint/">
  <FeatureManifests>
    <FeatureManifest Location="WFInspector\feature.xml" />
    <FeatureManifest Location="WFInspectorStaple\feature.xml"/>
  </FeatureManifests>
  <TemplateFiles>
    <TemplateFile Location="FEATURES\WFInspector\default.aspx"/>
  </TemplateFiles>
  <Assemblies>
    <Assembly DeploymentTarget="WebApplication"
      Location="WFInspectorWebParts.dll">
      <SafeControls>
```

```
        <SafeControl Assembly="WFInspectorWebParts, Version=1.0.0.0,
          Culture=neutral, PublicKeyToken=42c55593c85ca4ad"
          Namespace="WFInspectorWebParts" TypeName="*" Safe="True"/>
      </SafeControls>
    </Assembly>
  </Assemblies>
  <DwpFiles>
    <DwpFile FileName="WFInspectorWebParts.webpart"
      Location="WFInspectorWebParts.webpart" />
  </DwpFiles>
</Solution>
```

Creating the DDF File

Now that the *manifest.xml* file is complete, it's time to create the CAB file that is the solution package. To create the solution package, use the *MAKECAB.EXE* utility because the setup and deployment project in Visual Studio that makes CAB files can't make them with subdirectories. Solution file ubdirectories must match the locations referenced in *manifest.xml*. To get those subdirectories in the solution file, use a DDF that *MAKECAB* knows how to read. In that file you'll specify the compression options, the files to compress, and what subdirectories in the CAB file to place them in.

The basic format of a DDF file is that of a text file. It contains one command per line. A line beginning with a single period is considered to be a command. Generally the first command is *.OPTION EXPLICIT*. For those of you familiar with Visual Basic, you'll recognize this sequence of words, which in this case means that if there's a problem processing the file *MAKECAB* should exit and report the error. The other commands that you'll use at the top of every DDF file that you're using to create a solution are:

- *.Set CabinetNameTemplate* – Specifies the name of the output file—in this case, *WFInspector.wsp*
- *.Set DiskDirectoryTemplate=CDROM* – Indicates that all of the CAB goes into a single directory
- *.Set CompressionType=MSZIP* – Indicates that all of the files will be compressed into CAB files
- *.Set UniqueFiles="ON"* – Indicates that all of the files referenced must be unique
- *.Set Cabinet=On* – Use cabinet files
- *.Set DiskDirectory1=.* – Use the current directory for the output CAB file

With that part out of the way you can just start listing files until you need to change the directory in the CAB file. Because the project to create the solution was created as an empty project underneath the same Visual Studio solution tree, most of the references will include a double dot (up one directory) notation and then back down into the correct folder. In every DDF for a solution file it's a good idea to start with *manifest.xml* since you know it's required. In this example there are three files that are saved in the root directory of the solution file: *manifest.xml*, *WFInspectorWebParts.dll*, and *WFInspectorWebParts.webpart*. These three lines look like this:

```
Manifest.xml
..\WFInspectorWebParts\bin\release\WFInspectorWebParts.dll
..\WFInspectorWebParts\WFInspectorWebParts.webpart
```

Next, set the new CAB directory to *WFInspector* with *.Set DestinationDir=WFInspector line*, which adds the two feature files (*feature.xml* and *module.xml*), which are deployed through the <FeatureManifest> element. Then set the directory to *FEATURES\WFInspector* and include the *default.aspx* file. Deploying all three of the files into the same directory may seem odd. Why do they end up in two different folders in the solution file? Sadly, they do because they're deployed differently; the relative paths referenced in *Manifest.xml* are different, and they will land in the correct directory on the server.

The final step is to add the files for the *WFInspectorStaple* feature. The whole DDF file looks like this:

```
.OPTION EXPLICIT
.Set CabinetNameTemplate=WFInspector.wsp
.Set DiskDirectoryTemplate=CDROM
.Set CompressionType=MSZIP
.Set UniqueFiles="ON"
.Set Cabinet=on
.Set DiskDirectory1=.
manifest.xml
..\WFInspectorWebParts\bin\release\WFInspectorWebParts.dll
..\WFInspectorWebParts\WFInspectorWebParts.webpart
.Set DestinationDir=WFInspector
..\WFInspectorFeature\feature.xml
..\WFInspectorFeature\module.xml
.Set DestinationDir=FEATURES\WFInspector
..\WFInspectorFeature\default.aspx
.Set DestinationDir=WFInspectorStaple
..\WFInspectorStaple\feature.xml
..\WFInspectorStaple\element.xml
```

Build and Deploy the Solution File

All of the hard work is done. From here on out it's pretty simple. Building the solution file is simply a matter of running *MAKECAB /F WPInspector.ddf*. The result is the solution file *WFInspector.WSP*. To make sure that it was created correctly, rename it so that its extension is *.CAB* and double-click it. Microsoft Windows Explorer will open the CAB file and display all of the files stored in it. Once you're comfortable that the file was created correctly rename it so that its extension is WSP to be able to deploy it to the SharePoint farm.

Deploying the solution file to SharePoint is a two-step process: First, register the solution in the database. Registration is accomplished from the *STSADM* command-line utility. To add the *wfinspector.wsp* file, run the utility like this:

```
STSADM –o addsolution –filename WFInspector.wsp
```

Second, once the solution has been added to the solution store you can deploy it from the command line, or you can deploy it using the Central Administration user interface. Find the Solutions Management page by going to Start-->Administrative Tools-->SharePoint 3.0 Central Administration-->Operations, and in the Global Configuration section clicking Solution management. From there, click one of the listed solutions to see its status and to take action on it—like deploying it.

Alternatively, you can deploy the solution from the command line with the *STSADM* command *deploysolution*. If you want to deploy the *wfinspector.wsp* solution to the local system, issue this command:

```
STSADM –o deploysolution –name wfinspector.wsp –allcontenturls –local –
allowgacdeployment –allowcaspolicies
```

The *–allowgacdeployment* and *–allocaspolicies* options are necessary only if you have indicated deploying an assembly to the GAC, or you have provided a CAS policy as a part of the solution file, respectively.

Extra Credit: Deploy a Site Definition

While it's much more common to deploy features and web parts, you may occasionally need to deploy site definitions. In this scenario you'll need to include <SiteDefinitionManifests> and <SiteDefinitionManifest> nodes in your *manifest.xml* file (directly under the root <Solution> node). The *SiteDefinitionManifest* element contains a single attribute—*Location*—which in this case is the directory name for the site definition as it will appear under the *…12\TEMPLATE\SiteTemplates* directory.

Inside the <SiteDefinitionManifest> node is a <WebTempFil> node that also has only a *Location attribute*. This attribute refers to the relative path and name of the *Webtemp*.xml* file from the *TEMPLATE* directory. For instance, to deploy a site definition installed in the *DEVX* directory with a *webtemp*.xml* file named *WEBTEMP_DEVX.xml*, the XML fragment would be:

```
<SiteDefinitionManifests>
  <SiteDefinitionManifest Location="DEVX">
    <WebTempFile Location="1033\XML\WEBTEMP_DEVX.XML" />
  <SiteDefinitionManifest>
</SiteDefinitionManifests>
```

In the DDF file the site definitions files are in the directory that you referenced in the *SiteDefinitionManifest*'s *Location* attribute—in this case, *DEVX*—and below. Thus, the *ONET.XML* file would be in *DEVX\XML*. The *WebTemp* file is stored in a directory with a path matching the path used in the *Location* attribute of the <WebTempFile> node.

The true beauty of solutions files is that in the end they are easy to make, and they create a repeatable process for installing and uninstalling custom code. For those in larger environments, where change control is critical, Solutions are a great way to ensure that the same files are deployed and the same configuration changes are made in each environment.

About The Author

Robert Bogue, MCSE (NT4/W2K), MCSA:Security, A+, Network+, Server+, INet+, IT Project+, E-Biz+, CDIA+, is president of Thor Projects LLC, which provides SharePoint Consulting services to clients around the country. He has contributed to more than 100 book projects and numerous other publishing projects. His latest book is The SharePoint Shepherd's Guide for End Users. (You can find out more about the book at www.SharePointShepherd.com.)

Bogue has been part of the Microsoft Most Valuable Professional (MVP) program for the past 5 years. He was most recently awarded for Microsoft Office SharePoint Server. Before that, Bogue was a Microsoft Commerce Server MVP and Microsoft Windows Servers-Networking MVP.

Bogue runs the SharePoint Users Group of Indiana (SPIN, www.spindiana.com), and he is also a member of the steering committees for the Indiana Windows Users Group and Indianapolis .NET Developer Association. In addition to speaking at local and regional events, Bogue speaks at national conferences. He blogs at www.thorprojects.com/blog , and you can reach him at Rob.Bogue@thorprojects.com .