akumina

Akumina Digital Workplace

Deploy your custom site definition

Version 1.1 (May 2016)

Table of Contents

PREREQUISITES 4

AN INTRODUCTION TO THE DIGITAL WORKPLACE DEPLOYMENT 4

ADDING SITEPROVISIONING.SAMPLESITE TO INTERCHANGE 5

CUSTOM SITE DEFINITION OVERVIEW 6

The C# class inheriting the SiteProvisionerSiteBase class 6
SiteProvisionerSiteBase 8
ISiteProvisionerSite 8

C# classes that inherit the ISiteProvisioner Step interface 9
Overview 9
Example Step 11

SiteDefinitions Folder 13
Branding 13
ContentEditor 14
ContentTypes 14
ListDefinitions 14
MasterPages 14
PageContent 14
PageLayouts 14
Pages 14
UploadFiles 14

SAMPLE STEP CODE 15
SampleService.cs 15

Sample Step Code – No User Inputs 16
SampleStep.cs (Step) 16
Example.cs (Inherits SiteProvisionerSiteBase) 17

Sample Step Code – With User Inputs 19
SampleStepWithUserInputs.cs (Step) 19
Example.cs (Inherits SiteProvisionerSiteBase) 21

CUSTOM SITE DEFINITION XML 23
ContentTypes.xml 23
Lists.xml 24
Update.xml 26
pages.xml 26
PageLayouts – Elements.xml 27
Pages – Elements.xml 28

SUPPORTED TOKENS 29
List.xml & Update.xml 29

MasterPage 29
pages.xml 29

OOB SAMPLESITE STEP CLASSES 29

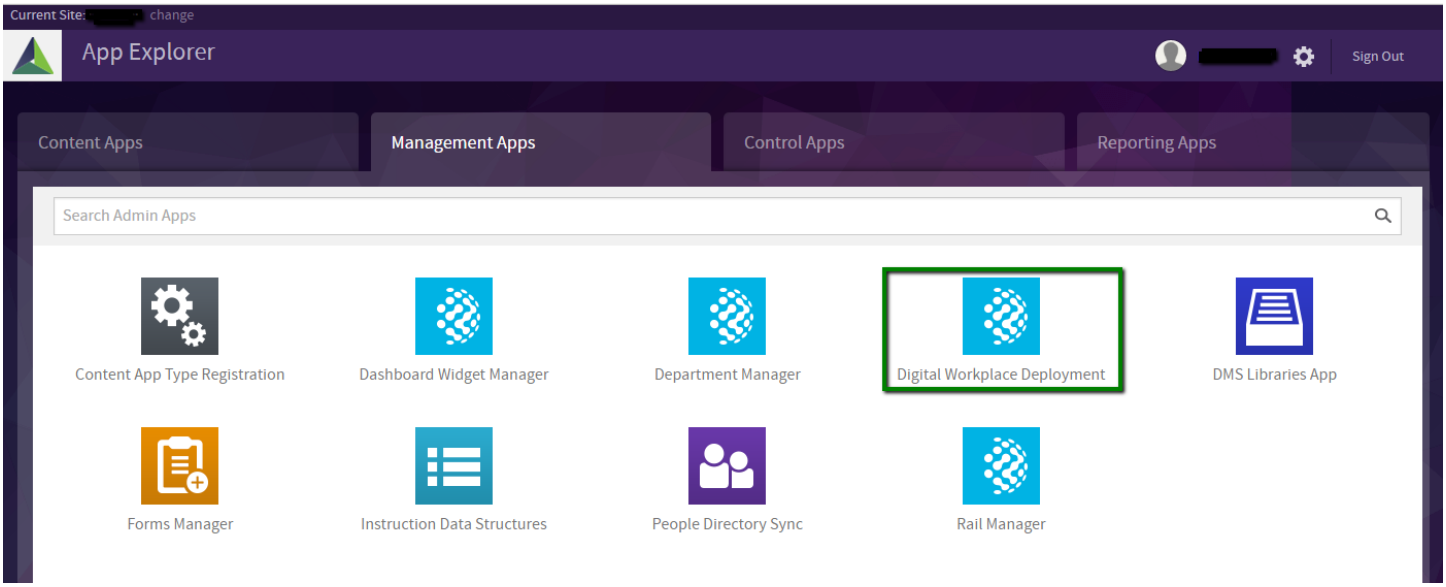
Prerequisites

- **A publishing site collection in Office 365**
- **Akumina InterChange (hosted on Azure) installed on Office 365**
- **InterChange Admin apps set accessible to a user group within the site**
- **The InterChangeSDK project**

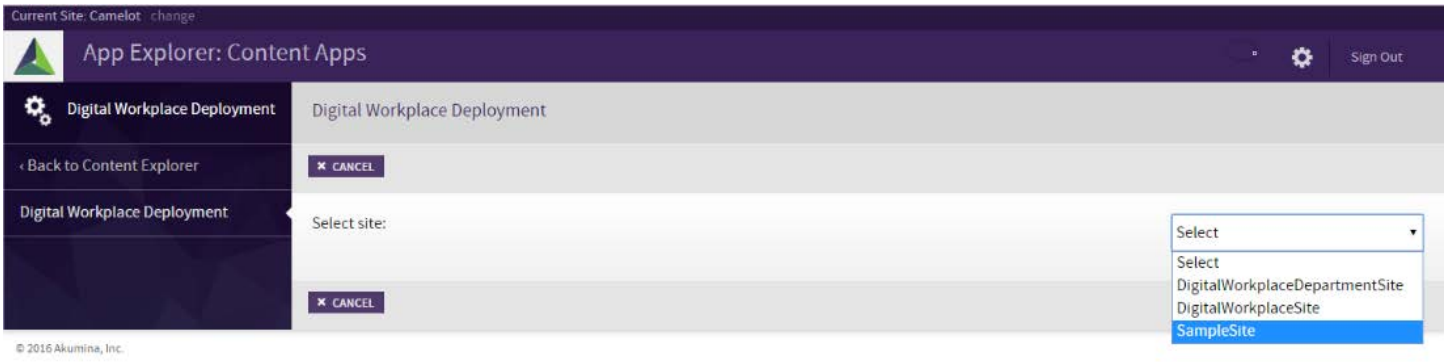
An Introduction to the Digital Workplace Deployment

Before getting started with the rest of the document, let's review the Digital Workplace Deployment UI.

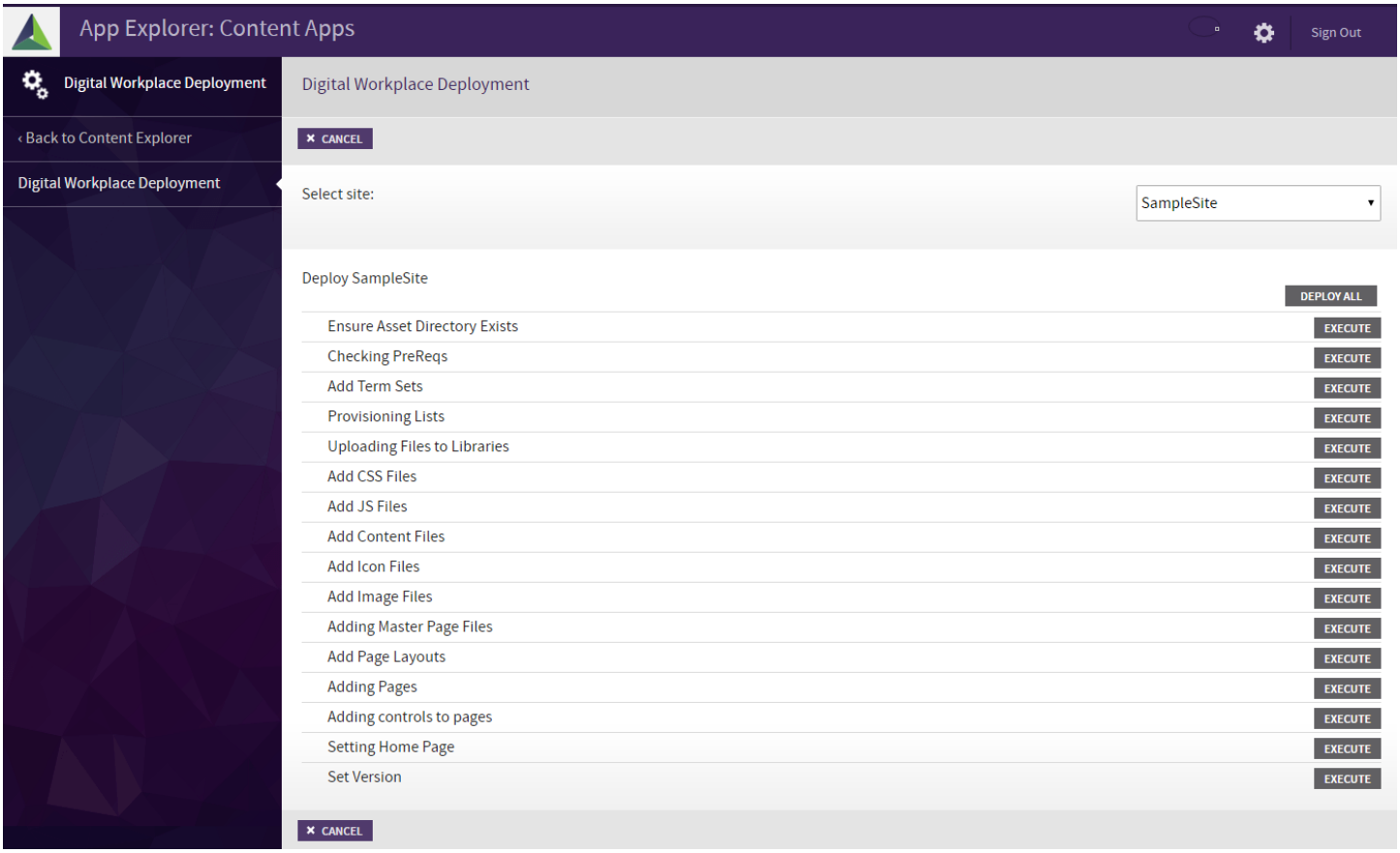
The Digital Workplace Deployment app can be found under the Management Apps tab in the InterChange explorer.



The Digital Workplace App allows us to select from the different site definitions we have created to deploy to the site.



Once we select a site definition we are given the option to execute all of the deployment steps (deploy all) or execute steps individually (execute).



Once we deploy all steps from our site collection, our branding objects will be deployed to the site. In example, the home page of our SharePoint site with some simple deployed branding is shown below.

Adding SiteProvisioning.SampleSite to InterChange

Within the InterChangeSDK project will be a project called SiteProvisioning.SampleSite. This is a basic example of a custom site definition we can deploy with the Digital Workplace Deployment App.

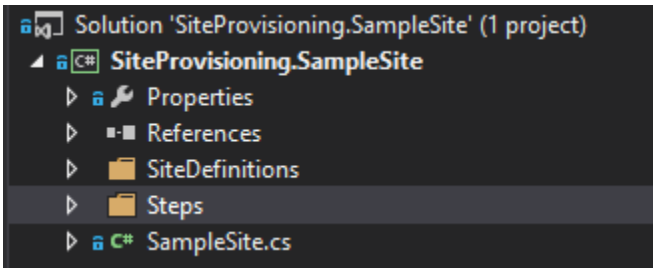
Adding the SiteProvisioning.SampleSite project with the Digital Workplace Deployment App can be done in a few simple steps.

- **Build the SiteProvisioning.SampleSite project**
- **Copy the SiteProvisioning.SampleSite.dll from $\InterChangeSDK\Main\Src\SiteProvisioning.SampleSite\bin\Debug over to Akumina.Interchange.Web\bin within your instance of InterChange (use FTP if you're hosting it on Azure).**
- **Copy the $\InterChangeSDK\Main\Src\SiteProvisioning.SampleSite\SiteDefinitions folder over to Akumina.Interchange.Web within your instance of InterChange**
- **Log into InterChange, go to the Digital Workplace Deployment App within the Management Apps tab. You should now see "SampleSite" appear in the dropdown.**

Any custom site definitions can be added to the Digital Workplace Deployment App in the same manner.

Custom Site Definition Overview

A Custom Site Definition will be contained within a C# project.



There are three elements within the project that are needed for your custom site definition to work with the Digital Workplace Deployment app

- **A C# class that inherits SiteProvisionerSiteBase**
- **C# classes that inherit the ISiteProvisionerStep interface (our steps)**
- **A SiteDefinitions folder**

We will go into each section in depth.

The C# class inheriting the SiteProvisionerSiteBase class

This file inherits the SiteProvisionerSiteBase class and assembles our step classes into a list of steps which is seen in the Digital Workplace Deployment App UI. This class is required for the site to show up in the Digital Workplace Deployment App dropdown. You may have multiple classes that inherit the SiteProvisionerSiteBase class within a csproj, they will all show up in the Digital Workplace Deployment App dropdown. A sample SiteDefinition.cs, called SampleSite.cs, is shown below

```
public class SampleSite : SiteProvisionerSiteBase
    {
        public override string AssetDirectory
        {
            get
```

```
            {
                return "SampleSite";
            }
        }

        public override string Javascript
        {
            get
            {
                  return “”;
            }
        }

        public override string FriendlyName
        {
            get
            {
                  return “Sample Site”;
            }
        }

        public override List<ISiteProvisionerStep> Steps
        {
            get
            {
                  var _steps = new List<ISiteProvisionerStep>();

                  // Ensure AssetDirectory Exists
                  _steps.Add(new EnsureAssetDirectoryExists());
                  // Ensure site is a Publishing Site
                  _steps.Add(new CheckPreReqs());
                  //      Do something with SampleStep step
                  _steps.Add(new SampleStep());

                  return _steps;
            }
         }
      public override List<SiteProvisionerSettingsField> UserSettings
        {
            get
```

```
            {
                return new List<SiteProvisionerSettingsField>();
            }
        }
    }
```

We'll cover this file by sections

public override string AssetDirectory

We set the AssetDirectory property here, which is used when some steps access the AssetDirectory inside of the SiteDefinitions folder. In this case, our Asset Directory is named SampleSite. Asset Directories will be discussed in depth under the *SiteDefinitions* section of the document.

public override string Javascript

The Javascript property can be used if you want to load javascript into the Digital Workplace Deployment App view. Set it to a blank string if you do not wish to load scripts to the view.

public override string FriendlyName

This is the name of the site definition that will show up within the drop down menu of the Digital Workplace Deployment App view.

public override List<ISiteProvisionerStep> Steps

Here is where we add our steps to the step list (so they show up in Digital Workplace Deployment App). Our step list must be of the type List<ISiteProvisionerStep>() and our entries are just instances of our classes that inherit the ISiteProvisionerStep interface. In this case we're adding the EnsureAssetDirectoryExists step, the CheckPreReqs step, and the Sample step. We will discuss step classes under the *C# classes that inherit the ISiteProvisioner Step interface* section.

public override List<SiteProvisionerSettingsField> UserSettings

Here is where we add user inputs to the Digital Workplace App. Our input list must be of the type List<SiteProvisionerSettingsField>() and our entries are instances of the SiteProvisionerSettingsField class. In this case, we have any user inputs so we return an empty List<SiteProvisionerSettingsField>() object. We will discuss adding user input fields in the *Sample Step Code – With User Inputs* section.

SiteProvisionerSiteBase

As we know, our C# class must inherit the SiteProvisionerSiteBase class. Simply put, the SiteProvisionerSiteBase class is an abstract class that fulfills the ISiteProvisionerSite interface and performs some backend work to allow our custom site definition and its steps show up in the Digital Workplace Deployment App.

```
public abstract class SiteProvisionerSiteBase : ISiteProvisionerSite
```

ISiteProvisionerSite

The SiteProvisionerSiteBase class implements the ISiteProvisionerSite interface, which is shown below.

```
public interface ISiteProvisionerSite
    {
        SiteProvisionerProperties Properties { get; set; }
```

```
        ApplicationContext Context { get; set; }
        string AssetDirectory { get; }
        string FriendlyName { get; }
        string Javascript { get; }
        List<ISiteProvisionerStep> Steps { get; }
        List<SiteProvisionerSettingsField> UserSettings { get; }
        SiteProvisionerResponse ExecuteSteps();
        SiteProvisionerResponse ExecuteStepByName(string stepName);
    }
```

C# classes that inherit the ISiteProvisioner Step interface

Overview

The Steps folder contains the classes that are executed by the Digital Workplace Deployer when we deploy our sites. When we created new steps they will be placed in this folder. Now this raises the question, what makes a regular old C# class a Digital Workplace Step? A step inherits the SiteProvisionStepBase class and implements the ISiteProvisionerStep interface.

```
public class SampleStep : SiteProvisionerStepBase, ISiteProvisionerStep
```

ISiteProvisionerStep

The Digital Workplace Deployment app is looking for a class to implement the ISiteProvisionerStep interface in order to be considered a step. The interface is shown below

```
public interface ISiteProvisionerStep
    {
        SiteProvisionerProperties Properties { get; set; }
        string StepName { get; }
        List<SiteProvisionerSettingsField> UserSettings { get; set; }

        SiteProvisionerStepResponse Execute();
    }
```

For the Properties property we make use of the SiteProvisionerProperties class to keep track of universal values for the site, which is shown below

```
public class SiteProvisionerProperties
    {
        public SiteProvisionerProperties();
```

```
        public string AssetDirectory { get; set; }
        public Dictionary<string, object> Values { get; set; }
    }
```

The SiteProvisionerSettingsField class to provide a format for defining fields for user input

```
public class SiteProvisionerSettingsField
    {
        public SiteProvisionerSettingsField();

        public object DefaultValue { get; set; }
        public string Description { get; set; }
        public string Name { get; set; }
        public bool Required { get; set; }
        public Type Type { get; set; }

        public bool Validate(object val);
    }
```

And the SiteProvisionerStepResponse class for the response from each step upon completion or error

```
public class SiteProvisionerStepResponse
    {
        public SiteProvisionerStepResponse();

        public bool Halt { get; set; }
        public string Message { get; set; }
        public bool Success { get; set; }
    }
```

SiteProvisionerStepBase

Fortunately, the ISiteProvisionerStep interface is completely implemented by the SiteProvisionerStepBase, which consequently allows us to simple override these values within our Step. We can see this in our Sample Step Deployment section. The SiteProvisionerStepBase is shown below

```
public abstract class SiteProvisionerStepBase : ISiteProvisionerStep
    {
        protected SiteProvisionerStepBase();

        public SiteProvisionerProperties Properties { get; set; }
        public abstract string StepName { get; }
        public List<SiteProvisionerSettingsField> UserSettings { get; set; }

        public abstract SiteProvisionerStepResponse Execute();
        protected string GetProperty(string name);
        protected string GetSiteId();
        protected int GetUnitOfWork();
        protected void SetSiteId(string siteId);
        protected void SetUnitOfWork(int unit);
    }
```

Example Step

Below we have the AddPages.cs step. Allow us to 'step' through it for a deeper exploration the step architecture.

```
using Akumina.SiteProvisioning.Common.Models;
using Akumina.SiteProvisioning.Core;
using Akumina.SiteProvisioning.Core.Interfaces;
using Akumina.SiteProvisioning.Core.Services;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SiteProvisioning.SampleSite.Steps
{
    public class AddPages : SiteProvisionerStepBase, ISiteProvisionerStep
    {
        public override string StepName
        {
            get
            {
                return "Adding Pages";
            }
```

```
        }

        public override SiteProvisionerStepResponse Execute()
        {
            var response = new SiteProvisionerStepResponse();

            PageService pageService = new PageService();

            try
            {
                pageService.AssetDirectory = this.Properties.AssetDirectory;
                pageService.SiteId = this.GetSiteId();

                pageService.AddPages();

                response.Message = "Pages Added";
                response.Success = true;
            }
            catch (Exception ex)
            {
                response.Message = ex.Message;
                response.Success = false;
            }

            return response;
        }
    }
}
```

We'll cover this file in sections

public override string StepName

We're setting the StepName property here to Adding Pages. The StepName is the name of the step in the Digital Workplace Deployment UI.

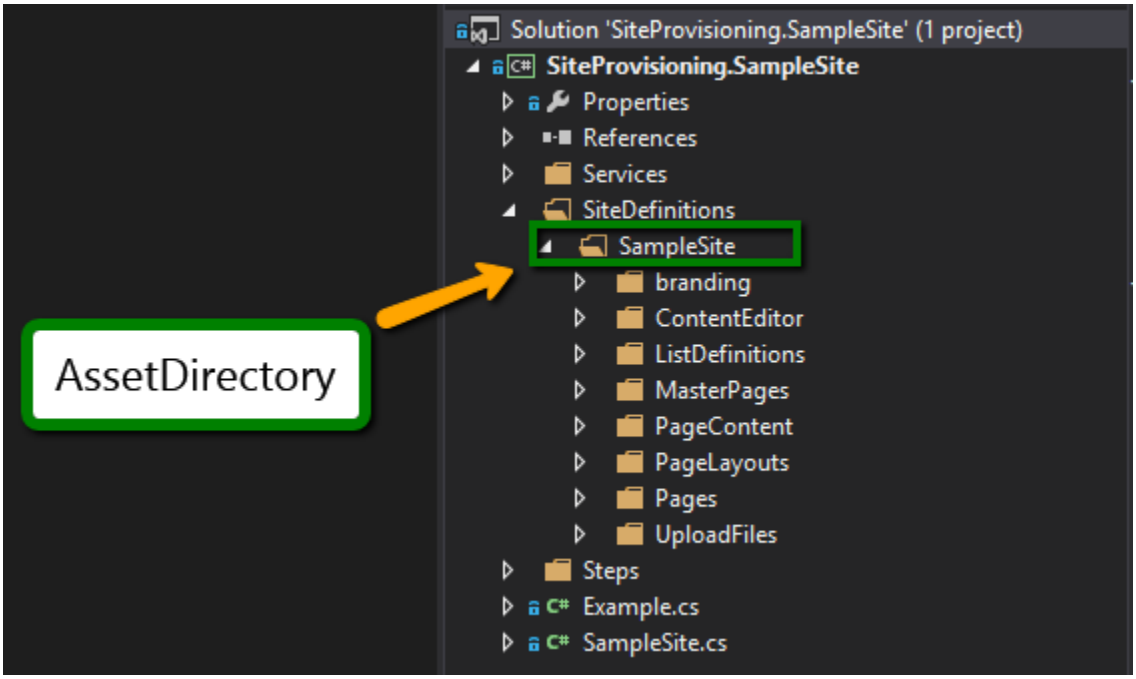
public override SiteProvisionerStepResponse Execute()

Here is where we execute the work done by the step itself. Our backend code is executed in the PageService class, it is always good practice to export the work done by the step to another class.

We return a SiteProvisionerStepResponse object from our Execute method. The response determines the success or fail message within the UI.

SiteDefinitions Folder

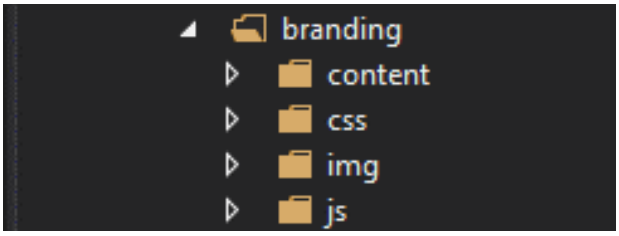
The Site Definitions folder contains our AssetDirectory folders. Each AssetDirectory folder contains all assets needed for the deployment of the respective site (branding files, layouts, xml files). The name of our AssetDirectory is important as we will be referring to it within all of our deployment steps when referencing these files. The Site Definitions folder may have multiple Asset Directories.



Within the AssetDirectory folder is the following framework.

Branding

The Branding folder contains all files that will be provisioned into the Style Library list upon deployment. The directories listed below will maintain their structure under the AssetDirectory folder within the Style Library when utilizing the OOB steps we provide with SampleSite deployer. Any files we want deployed to the Style Library should be placed within one of the listed directories.



content

Store your Mustache templates within the templates folder inside of the content directory

css

Store custom css for the site definition within the CSS folder.

img

Store images in the img folder

js

Store custom js for the site definition within the js folder.

ContentEditor

The ContentEditor folder contains the ContentEditor.xml file, we need this for deploying our widgets. **Do not edit or delete this file**.

ContentTypes

The ContentTypes folder contains information that the Digital Workplace Deployment App uses to create content types on a site. This information is stored in an xml file called ContentTypes.xml. Edit this file to add addition content types to the site. We will discuss how to do so in the *Custom Site Definition XML* section.

ListDefinitions

The ListDefinitions folder contains information that the Digital Workplace Deployment App uses to create lists on a site. This information is stored in an xml file called Lists.xml. Edit this file to add additional lists to the site. We will discuss how to do so in the *Custom Site Definition XML* section

MasterPages

The MasterPages folder will contain a single html file. The html file is deployed to the site's Master Page Gallery, an associated .master file is generated, and the master page is set as the site master page with the AddMasterPageFiles.cs step. **There can only be one html file in this directory for the masterpage deployment**. This will need to be in the proper html master page format in order for the deployment to work, otherwise SharePoint will throw errors. Use the samplemaster.html in the SiteProvisioning.SampleSite as a model for a simple master page. For more info on creating Master Page html, see https://msdn.microsoft.com/en-us/library/office/jj822370.aspx.

PageContent

The PageContent folder contains information that the Digital Workplace Deployment App will use to add content to the pages we create. This information is stored within an xml file called pages.xml. Edit this file to specify content markup you want on each page. We will discuss how to do so in the *Custom Site Definition XML* section.

PageLayouts

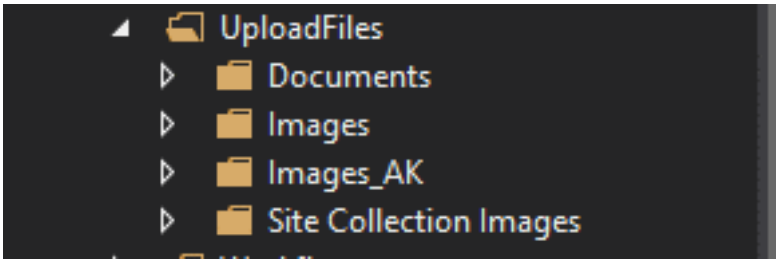
The PageLayouts folder is where we store the page layouts we want deployed to our site's Master Page Gallery by the AddPageLayouts.cs step. Inside of the file will be our page layouts (.aspx files) and an Elements.xml file, which informs the deployer of what layouts to deploy. When you add a page layout to this folder, you will need to edit the Elements.xml file accordingly. We will discuss how to do so in the *Custom Site Definition XML* section.

Pages

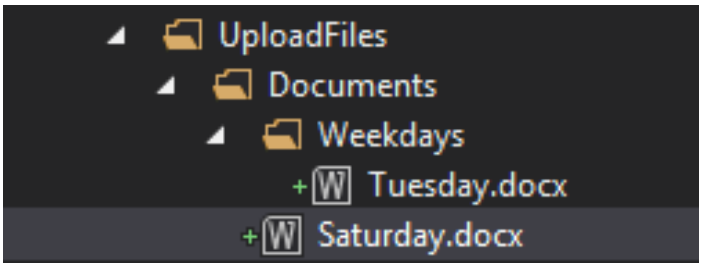
The Pages folder designates the pages that are deployed to the Pages Library of our site. This information is stored within an xml file called Elements.xml. Edit this file to add additional pages to the site. We will discuss how to do so in the *Custom Site Definition XML* section.

UploadFiles

The UploadFiles folder contains all files that we want uploaded to SharePoint file libraries within our site via the UploadFiles.cs step. We can designate the library that we want our files uploaded to by the name of the subdirectory under our UploadFiles folder. In the example below, we are uploading files into the Documents, Images, Images_AK, and Site Collection Images libraries.



The UploadFiles.cs step treats these subdirectories as the libraries themselves and copies the file structure underneath into the corresponding SharePoint library. For example, if we wanted to upload the Saturday.docx into the Documents library and the Tuesday.docx into a folder called Weekdays inside of the document library, we would arrange our file structure as shown below



Sample Step Code

Here are some samples

SampleService.cs

We will utilize SampleService as the backend for our Sample Steps

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SiteProvisioning.SampleSite.Services
{
    public class SampleService
    {
        public SampleService()
        {

        }

        public void DoSomething()
```

```
        {
            //Wait 3 seconds
            System.Threading.Thread.Sleep(3000);
        }

        public string DoSomethingWithTitle(string title)
        {
            //Wait 2 seconds
            System.Threading.Thread.Sleep(2000);

            //Return Title
            return title;
        }
    }
}
```

Sample Step Code – No User Inputs

SampleStep.cs (Step)

A step that calls one method in SampleService and completes.

```
using Akumina.SiteProvisioning.Common.Models;
using Akumina.SiteProvisioning.Core;
using Akumina.SiteProvisioning.Core.Interfaces;
using Akumina.SiteProvisioning.Core.Services;
using SiteProvisioning.SampleSite.Services;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SiteProvisioning.SampleSite.Steps
{
    public class SampleStep : SiteProvisionerStepBase, ISiteProvisionerStep
    {
        public override string StepName
        {
            get { return "Executing a Sample Step"; }
        }
```

```
        public override SiteProvisionerStepResponse Execute()
        {
            var response = new SiteProvisionerStepResponse();

            SampleService sampleService = new SampleService();
            sampleService.AssetDirectory = this.Properties.AssetDirectory;

            try
            {
                sampleService.DoSomething();

                response.Message = "We did something";
                response.Success = true;
            }
            catch (Exception ex)
            {
                response.Message = ex.Message;
                response.Success = false;
            }

            return response;
        }
    }
}
```

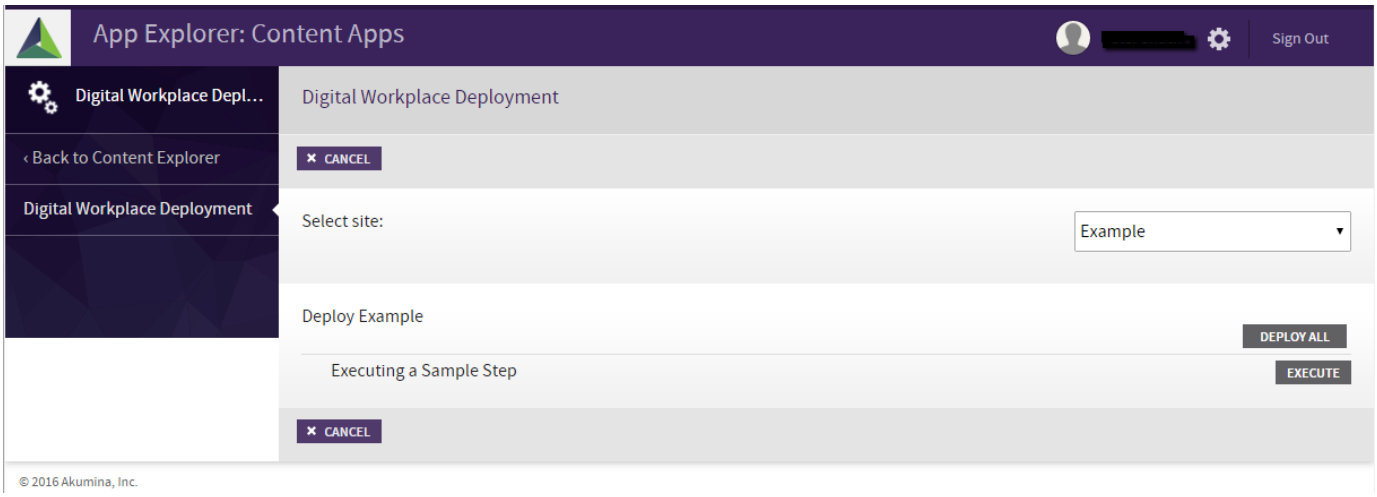
Example.cs (Inherits SiteProvisionerSiteBase)

Our site deployment will consist of one step, SampleStep.cs

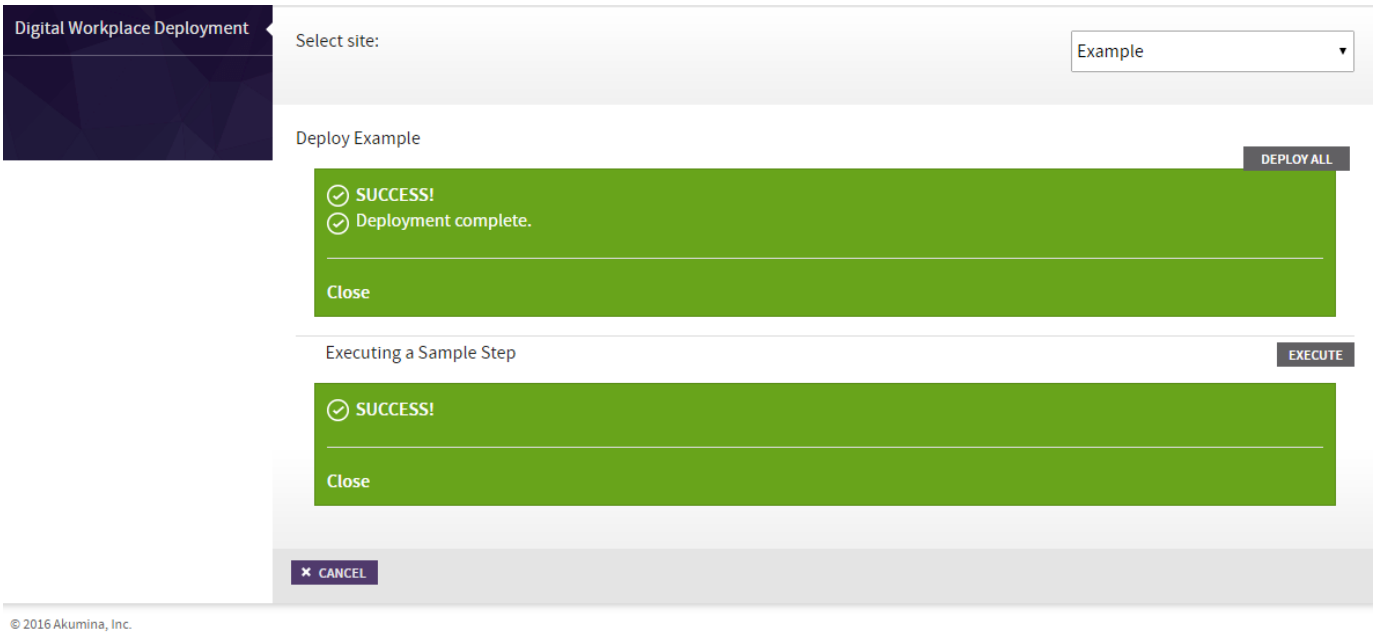
```
using Akumina.SiteProvisioning.Core.Interfaces;
using SiteProvisioning.SampleSite.Steps;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SiteProvisioning.SampleSite
{
    public class Example : SiteProvisionerSiteBase
```

```
    {
        public override string AssetDirectory
        {
            get { return "Example"; }
        }

        public override string Javascript
        {
            get { return “”; }
        }

        public override string FriendlyName
        {
            get { return “Example Site”; }
        }

        public override List<ISiteProvisionerStep> Steps
        {
            get
            {
                var _steps = new List<ISiteProvisionerStep>();

                _steps.Add(new SampleStep());

                return _steps;
            }
        }

        public override List<SiteProvisionerSettingsField> UserSettings
        {
            get
            {
                return new List<SiteProvisionerSettingsField>();
            }
        }
    }
}
```

Once you integrate your custom site definition with the Digital Workplace Deployment App you should be able to see it in the UI



And when you click Deploy All you will see the response message.



Sample Step Code – With User Inputs

For this example we will reuse the SampleService.cs class from the previous example.

SampleStepWithUserInputs.cs (Step)

Our new step will utilize a "Title" field for user input

```
using Akumina.SiteProvisioning.Common.Models;
using Akumina.SiteProvisioning.Core;
using Akumina.SiteProvisioning.Core.Interfaces;
using Akumina.SiteProvisioning.Core.Services;
using SiteProvisioning.SampleSite.Services;
```

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SiteProvisioning.SampleSite.Steps
{
    public class SampleStepWithUserInputs : SiteProvisionerStepBase, ISiteProvisionerStep
    {
        public override string StepName
        {
            get { return "Executing a Sample Step with User Inputs"; }
        }

        public override SiteProvisionerStepResponse Execute()
        {
            var response = new SiteProvisionerStepResponse();

            SampleService sampleService = new SampleService();
            sampleService.AssetDirectory = this.Properties.AssetDirectory;

            try
            {
                var siteTitle = GetProperty("Title");

                if (string.IsNullOrEmpty(siteTitle))
                {
                    throw new Exception("The Title value is required");
                }
                response.Message = sampleService.DoSomethingWithTitle(siteTitle);
                response.Success = true;
            }
            catch (Exception ex)
            {
                response.Message = ex.Message;
                response.Success = false;
            }

            return response;
```

```
        }
    }
}
```

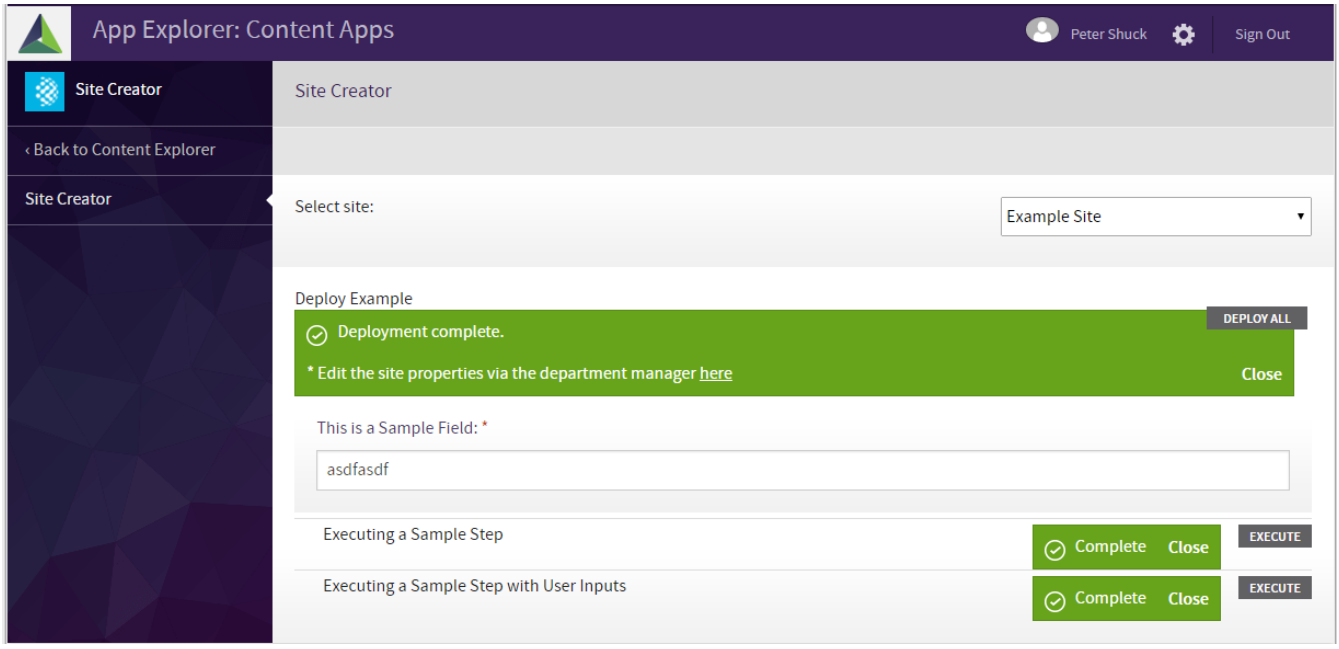
Example.cs (Inherits SiteProvisionerSiteBase)

We will reuse Example.cs from the previous sample, but with a couple modifications. We will add our SampleStepWithUserInputs class to the step list and add an entry to the SiteProvisionerSettingsField list. Changes from the previous version are highlighted.

```
using Akumina.Logging;
using Akumina.SiteProvisioning.Common.Models;
using Akumina.SiteProvisioning.Core;
using Akumina.SiteProvisioning.Core.Interfaces;
using SiteProvisioning.SampleSite.Steps;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace SiteProvisioning.SampleSite
{
    public class Example : SiteProvisionerSiteBase
    {
        public override string AssetDirectory
        {
            get { return "Example"; }
        }

        public override string Javascript
        {
            get { return “”; }
        }

        public override string FriendlyName
        {
            get { return “Example Site”; }
        }

        public override List<ISiteProvisionerStep> Steps
```

```
        {
            get
            {
                var _steps = new List<ISiteProvisionerStep>();

                _steps.Add(new SampleStep());

                _steps.Add(new SampleStepWithUserInputs());

                return _steps;
            }
        }

        public override List<SiteProvisionerSettingsField> UserSettings
        {
            get
            {
                var _userSettings = new List<SiteProvisionerSettingsField>();

                _userSettings.Add(
                    new SiteProvisionerSettingsField()
                    {
                        Name = "Title",
                        Description = "This is a Sample Field",
                        DefaultValue = "",
                        Required = true
                    }
                );

                return _userSettings;
            }
        }
    }
}
```

After we integrate our project with the Digital Workplace App, we should see our steps and user input appear when we select the Example site within the app.



Custom Site Definition XML

We use xml files throughout our SampleSite to provide information about our site to the Digital Workplace Deployment App. Here we will go into detail about customizing them.

ContentTypes.xml

We define the Content Types we want added to our site in the ContentTypes.xml file within the ContentTypes folder. Below is a sample ContentTypes.xml

```
<lists>
<ContentType ID="0x01200200C25A08D46BE4487A86080142FBFED933"
Name="AkuminaDiscussionBoardContentType" Group="Akumina Content Types"
Description="Akumina Discussion Board Content Type" Inherits="TRUE" Version="0">
    <FieldRefs >
      <Field ID="{A6828C7D-6754-449A-AB5F-07105080D991}" Name=”AttachmentLinks”
Type="Note" DisplayName="AttachmentLinks" Group="Akumina Content Types" Required="FALSE"
ShowInEditForm="TRUE" ShowInNewForm="TRUE" />
      <Field ID="{2D33BCF0-F6B2-453D-97CD-BD3268967983}" Name=”ConfirmArchive”
Type="Boolean" DisplayName="ConfirmArchive" Group="Akumina Content Types"
Required="FALSE" ShowInEditForm="TRUE" ShowInNewForm="TRUE" />
    </FieldRefs>
  </ContentType>
</lists>
```

Within the ContentType tag we have multiple attributes of note. The ID attribute is not only the unique identifier for the content type, but it the starting digits are a direct reference to a parent content type. In this case, we are inheriting from 0x012002, which is the SharePoint Discussion content type. The additional digits are random and are used for unique identification. The Name attribute designates the name of the content type. The Group attribute designates the group that the content type belongs to. The Description attribute designates the description associated with the content type.

The Inherits attribute designates whether the content type inherits fields from its parent content type. The Version attribute designates the version number.

We nest the fields we are assigning to the content type within the FieldRefs tag. Each individual field is declared with a Field tag and its attributes determine its properties. The ID attribute is a unique guid we assign to the field for identification. The Name attribute designates the internal name of the field. The Type designates the data type that we are assigning to the field. For more info on data types see the ListDefinitions section. The DisplayName attribute designates the name that will appear in the content type view in SharePoint. The Group attribute designates the group we are assigning the field to. The Required field designates whether or not SharePoint will require data to be entered into that field when an item is created in a list inheriting this content type. The ShowInEditForm attribute designates whether or not the field will appear when we edit list items in a list that inherits this content type. The ShowInNewForm designates whether or not the field will appear when we create new list items in a list that inherits this content type. The Fields will be stored as site columns underneath their assigned group.

Lists.xml

We define the lists we want added to the site in the Lists.xml file within the ListDefinitions folder. Below is a sample Lists.xml file

```
<lists>
 <list name="SampleList_AK" templateID="104" contentType=”Akumina Sample Type”
Enforce="TRUE" noCrawl="TRUE">
    <Field Name="Start_x0020_Date"  DisplayName="Start Date" Type="DateTime"
Required="TRUE">
      <Default>[today]</Default>
    </Field>
    <Field Name="AnnouncementTitle"  DisplayName="AnnouncementTitle" Type="Text"
Required="TRUE" />
    <Field Name="Image" DisplayName="Image" Type="URL" Format="Image" Required="FALSE" />
    <Data>
      <Rows>
        <Row>
          <Field Name="Title">Rogers, Sherk and Maffei taps new leadership in
Nashua</Field>
          <Field Name="AnnouncementTitle">Akumina taps new leadership in Nashua</Field>
          <Field Name="Image">{SiteUrl}/Images_AK/Office.jpg</Field>
          <Field Name="Start_x0020_Date">8/22/2016</Field>
       </Row>
      </Rows>
    </Data>
  </list>
</lists>
```

In the sample we define one list. Within the list tag, the name attribute is the name of the SharePoint list to be created. The templateID property value is the list template used, in this case it is an announcement list (104). All list ids can be found at https://msdn.microsoft.com/en-us/library/microsoft.sharepoint.splisttemplatetype.aspx. The contentType attribute specifies the content type the list inherits from. If the list has a content type, it add the content type's columns to the list. The Enforce attribute determines whether or not the Title field within the list must have unique values. The noCrawl attribute sets the NoCrawl property of the list.

```
<list name="SampleList_AK" templateID="104" contentType=”Akumina Sample Type”
Enforce="TRUE" noCrawl="TRUE">
```

When we define fields we have multiple attributes to take into account. The Name attribute is what the field will be referred to internally, we will use the Name when adding data to the field. The DisplayName attribute is the name of the field will be in the List View. The Type attribute is the datatype of the field. The Required attribute is whether or not the field requires data when a new list item is created. A default value can be assigned to the field by included the <Default> tag. All list attributes can be found at https://msdn.microsoft.com/en-us/library/office/aa979575.aspx.

```
<Field Name="Start_x0020_Date"  DisplayName="Start Date" Type="DateTime" Required="TRUE">
      <Default>[today]</Default>
    </Field>
```

We define default data within the <Data><Rows> tags. Each list item will be nested within a <Row> tag and data will be added to the necessary fields. When adding a data to a field we reference the field by its Name attribute.

```
<Data>
      <Rows>
        <Row>
          <Field Name="Title">Rogers, Sherk and Maffei taps new leadership in
Nashua</Field>
          <Field Name="AnnouncementTitle">Akumina taps new leadership in Nashua</Field>
          <Field Name="Image">{SiteUrl}/Images_AK/Office.jpg</Field>
          <Field Name="Start_x0020_Date">8/22/2016</Field>
       </Row>
      </Rows>
    </Data>
```

Note how we also use the **{SiteUrl}** token within one of our fields values above. The **{SiteUrl}** token is replaced with the SharePoint site url when we deploy our lists. We will go into our supported tokens in more detail within the *Supported Tokens* section.

Update.xml

The Update.xml indicates items that need to be updated or provisioned within lists after the initial list provisioning. The Update.xml file should be stored within the ListDefinitions folder. The xml format is essentially the same as the List.xml with a couple exceptions. The Update attribute within the row tag designates if the item should be overwritten if it already exists within the list. We also will only designate data within the Update.xml, no additional lists or columns to lists can be defined in here. An example Update.xml is shown below.

```
<lists>
  <list name="Forms_AK">
    <Data>
      <Rows>
        <Row Update ="FALSE">
          <Field Name="Title">IT Request</Field>
          <Field Name="SiteId">{SiteId}</Field>
          <Field Name="ReferenceList">ITRequestAK</Field>
          <Field Name="ColumnOrder">first,last,email,location,service-
category,comments</Field>
          <Field Name="FormDescription">IT Request description</Field>
          <Field Name="ReferenceListId">{ReferenceListId:ITRequestAK}</Field>
         </Row>
      </Rows>
    </Data>
  </list>
</lists>
```

pages.xml

The PageContent folder will contain the pages.xml file. The pages.xml file contains the markup that will be deployed to the pages by the AddControlsToPages.cs step. A sample pages.xml file is listed below.

```
      <pages xmlns:ak="http://www.w3.org/ak">
        <page name="Home.aspx">
            <zone name="WebPartZone1">
                  <contenteditor>
              <h2>Sample home page</h2>
                  </contenteditor>
            </zone>
            <zone name="WebPartZone2">
                  <contenteditor>
            <div class="ak-controls ak-genericlist-widget" id="{{NewGuid}}"
ak:listname="" ak:displaytemplateurl="{{SiteUrl}}/Style
Library/digitalworkplace/Content/Templates/GenericList/DepartmentList.html"
ak:callbackmethod="ShowDepartmentItems" ak:callbacktype="customdataload"> </div>
                  </contenteditor>
```

```
        </zone>
      </page>
    </pages>
```

We designate which page that we're adding content to with the **name** attribute of the pages tag. This name must correspond to the name of a page we are provisioning.

We designate the webpart zone that we want the content deployed to with the **name** attribute of the zone tag. The name must correspond to the name of the WebPartZone within the page layout that the page inherits from.

The contenteditor tag signifies that we want the Content Editor webpart xml deployed to the page. Any content within the contenteditor tag will be added as inner markup to the Content editor webpart. Content can be simple markup as shown below:

```
<h2>Sample home page</h2>
```

Or it can be an Akumina widget (see the Akumina Digital Workplace Developer Guide for more info on widgets).

```
<div class="ak-controls ak-genericlist-widget" id="{{NewGuid}}" ak:listname=""
ak:displaytemplateurl="{{SiteUrl}}/Style
Library/digitalworkplace/Content/Templates/GenericList/DepartmentList.html"
ak:callbackmethod="ShowDepartmentItems" ak:callbacktype="customdataload">
</div>
```

Additionally, multiple Content Editor Web Parts can be placed within the same WebPartZone

```
<zone name="WebPartZone1">
      <contenteditor>
            <h2>Sample Multiple Content Editors</h2>
      </contenteditor>
      <contenteditor>
            <div class="ak-controls ak-genericlist-widget" id="{{NewGuid}}"
ak:listname="" ak:displaytemplateurl="{{SiteUrl}}/Style
Library/digitalworkplace/Content/Templates/GenericList/DepartmentList.html"
ak:callbackmethod="ShowDepartmentItems" ak:callbacktype="customdataload"> </div>
      </contenteditor>
</zone>
```

PageLayouts – Elements.xml

When we add new .aspx files to the PageLayouts folder, we need to customize the Elements.xml file, which informs the Digital Workplace Deployment App of what layouts to deploy. Below is a sample Elements.xml file.

```
<pagelayouts>
```

```
  <pagelayout name="DigitalWorkspace1ColumnPageLayout.aspx" folder="" title="Digital
Workspace 1 Column Page Layout"
publishingAssociatedContentType=";#AkuminaIgnite;#0x010100C568DB52D9D0A14D9B2FDCC96666E9F
2007948130EC3DB064584E219954237AF39002C646921DEB14195B1E318F903889C55;#" />
  <pagelayout name="DigitalWorkspace2ColumnLeftPageLayout.aspx" folder="" title="Digital
Workspace 2 Column Left Page Layout"
publishingAssociatedContentType=";#AkuminaIgnite;#0x010100C568DB52D9D0A14D9B2FDCC96666E9F
2007948130EC3DB064584E219954237AF39002C646921DEB14195B1E318F903889C55;#" />
  <pagelayout name="DigitalWorkspaceDashboardPageLayout.aspx" folder="" title="Digital
Workspace Dashboard Layout"
publishingAssociatedContentType=";#AkuminaIgnite;#0x010100C568DB52D9D0A14D9B2FDCC96666E9F
2007948130EC3DB064584E219954237AF39002C646921DEB14195B1E318F903889C55;#" />
</pagelayouts>
```

We define a page layout and its properties within the page layout tag. The name attribute is the file name of the page layout file, ie DigitalWorkspace1ColumnPageLayout.aspx. The title attribute is the display name, ie Digital Workspace 1 Column Page Layout. The folder attribute indicates the folder that the layout is stored in within the Masterpage Gallery. The publishingAssociatedContentType is the content type associated with the page layout. The value set to this attribute must be of the following format:

```
=";#<ContentTypeName>;#<ContentTypeId>;#"
```

The ContentTypeId will be in hexadecimal format.

However, we automatically create the AkuminaIgnite page layout content type with the AddPageLayouts.cs step, so assigning the publishingAssociatedContentType of new page layouts to be AkuminaIgnite will work just fine.

```
publishingAssociatedContentType=";#AkuminaIgnite;#0x010100C568DB52D9D0A14D9B2FDCC96666E9F
2007948130EC3DB064584E219954237AF39002C646921DEB14195B1E318F903889C55;#"
```

After we add a new page layout to the file we need to add an associated entry to the Elements.xml file. For example, if we add the layout SampleLayout.aspx we will add the following entry to our Elements.xml.

```
<pagelayout name="SampleLayout.aspx" folder="" title="Sample Page Layout"
publishingAssociatedContentType=";#AkuminaIgnite;#0x010100C568DB52D9D0A14D9B2FDCC96666E9F
2007948130EC3DB064584E219954237AF39002C646921DEB14195B1E318F903889C55;#" />
```

Pages – Elements.xml

The Pages folder contains an Elements.xml file that designates that pages that are deployed to the Pages Library. An example Elements.xml file that adds 3 pages to the site is shown below:

```
<pages>
  <page name="CompanyCalendar.aspx" layout="DigitalWorkspace1ColumnPageLayout.aspx"
title="Company Calendar Page"/>
  <page name="Dashboard.aspx" layout="DigitalWorkspaceDashboardPageLayout.aspx" title="My
Dashboard Page"/>
```

```
  <page name="Departments.aspx" layout="DigitalWorkspace1ColumnPageLayout.aspx"
title="Department Listing Page"/>
</pages>
```

A Page is defined within the page tag. The name attribute is the file name of the page (this will be a aspx file). The layout attribute is the page layout that the page is inheriting. The title attribute is the display name. Now if we wanted to create a Sample.aspx page using our SampleLayout.aspx described in the previous section, we would add the following entry to our Elements.xml

```
<page name="Sample.aspx" layout="SampleLayout.aspx" title="Sample Page"/>
```

Supported Tokens

Within our assets we support multiple tokens which are replaced with site specific values upon deployment. They are as follows.

List.xml & Update.xml

Tokens that we support within the List.xml and Update.xml.

- **{ListId} – the guid identifying the list**
- **{SiteId} – the guid identifying the site**
- **{SiteTitle} – the title of the site**
- **{SiteUrl} – the url of the site**
- **{SiteCollectionId} – the guid identifying the root site collection**
- **{SiteCollectionTitle} – the title of the root site collection**
- **{SiteCollectionUrl} – the url of the root site collection**
- **{ReferenceListId:ListName} – the guid identifying the list with the title ListName**

MasterPage

Tokens that we support within the masterpage.html file

- **{NewGuid} – a random guid, used for creating random identifiers**
- **{SiteUrl} – the url of the site**

pages.xml

Tokens that we support within the pages.xml

- **{SiteTitle} – The title of the current site**
- **{SiteUrl} – The url of the current site**
- **{SiteCollectionUrl} – The url of the root site collection**
- **{NewGuid} – A random guid, used for creating random identifiers**

OOB SampleSite Step Classes

We include the following classes that implement the ISiteProvisionerStep interface

- **AddContentFilesToStyleLibrary.cs – Adds the contents of Branding/Content to AssetDirectory/Content inside of the Style Library**
- **AddControlsToPages.cs – Adds the content outlined in PageContent/pages.xml to the designated web part zones**
- **AddCSSFilesToStyleLibrary.cs – Adds the contents of Branding/CSS to AssetDirectory/CSS within the Style Library**
- **AddImageFilesToStyleLibrary.cs – Adds the contents of Branding/img to AssetDirectory/img within the Style Library**
- **AddJSFilesToStyleLibrary.cs – Adds the contents of Branding/JS to AssetDirectory/JS within the Style Library**
- **AddMasterPageFiles.cs – Adds the master page html file from the MasterPages folder to the Master Page gallery on the site, generates an associated .master file, and sets the Site Master Page to the .master file.**
- **AddPageLayouts.cs – Adds the AkuminaIgnite content type to the site. Adds Page Layouts to the site.**
- **AddPages.cs – Adds pages to the Pages library**
- **CheckPrereqs.cs – Checks to see if Publishing features are activated on the SharePoint site.**
- **EnsureAssetDirectoryExists.cs – Ensures that the AssetDirectory exists with the SiteDefinitions folder**
- **ProvisionLists.cs – Adds lists to the site**
- **SampleStep.cs – Our Sample step without user inputs**
- **SampleStepWithUserInputs.cs – Our Sample step with user inputs**
- **SetHomePage.cs – Sets the default landing page of the site to Home.aspx**
- **UploadFiles.cs – Uploads files to SharePoint libraries**