**On Premise to Office 365 Migration**

Legacy SharePoint Development Approaches

You may find yourself wondering exactly how things got to this point. Many organizations using SharePoint today maintain large portfolios of custom code solutions that leverage different legacy development approaches advocated by Microsoft through the years. Beginning with SharePoint 2007, the server-side object model has given .NET developers the ability to build custom farm solutions on top of the platform, by providing classes that correspond to every building block of SharePoint, from the farm itself, all the way down to individual items in a single list. Farm solutions in the Feature framework were introduced in SharePoint 2007, giving developers the ability to package and deploy custom functionality, and applications that integrate seamlessly with the SharePoint platform. In the years that followed, organizations invested billions of dollars developing and maintaining custom solutions that do everything from modifying the look and feel of the SharePoint user interface to not look like SharePoint, to automating complex back-end processes through custom code. Much of SharePoint's popularity today can be attributed to the ease with which Microsoft has allowed it to be customized. This was all well and good, but with the great power afforded by farm solutions in the server object model came great responsibility, responsibility which many developers and organizations were ill prepared to take on. You see, farm solutions in SharePoint deploy assemblies to the global assembly cache, and run all code with full trust in the same IIS worker process as your SharePoint web applications. This can be a very risky proposition if code in a farm solution contains memory leaks or errors, as one rogue farm solution can potentially destabilize or even bring down the entire farm. I can attest to many occasions where the server administrators had to be called in the middle of the night to address outages that were caused by poorly-coded farm solutions. Furthermore, because farm solutions run code in the same IIS worker process as SharePoint, any deployment of a farm solution, whether a brand-new solution or an update to an existing one, requires a reset of the IIS application pools associated with the web applications where the solution is deployed. To make matters worse, farm solutions have to be deployed by a SharePoint farm administrator with console access to the server. This forced organizations to implement strict governance around when farm solutions could be deployed, generally during a small after-hours window, and made it very difficult to deploy small, incremental updates to deployed capabilities without introducing downtime that inconvenienced users and disrupted operations. To that end, and in an attempt to address the various shortcomings of farm solutions, Microsoft introduced sandboxed solutions in SharePoint 2010. Sandboxed solutions were intended to mitigate many of the inherent risks of farm solutions by providing process isolation for custom managed code. Instead of full trust code running in the same IIS worker process as SharePoint, code would run in a separate user code solution worker process called SPUCWorkerProcess.exe. Sandboxed solutions could be uploaded and activated by site collection administrators in a solutions gallery, and any deployment of a sandboxed solution did not require either the user code solution worker process or the SharePoint IIS worker process to be reset. This made deployments far easier and more transparent to end users. To prevent a sandboxed solution from consuming too many server resources while running, resource quotas could be implemented. It seemed like sandboxed solutions represented the best of all worlds. Unfortunately the tradeoffs required to make sandboxed solutions secure resulted in a severely limited set of capabilities for developers looking to write server-side code. Only a very limited subset of the SharePoint server object model could be used in sandboxed solutions. Most developers found that their requirements could not be met with sandboxed solutions, and they continued to build farm solutions to meet their customers' needs. In 2014, Microsoft officially deprecated the use of custom managed code in sandboxed solutions. As of 2016, code-based sandboxed solutions can no longer be deployed to SharePoint Online at all. Unfortunately, even many so-called no-code sandboxed solutions, which are those that do not contain any server-side code and consist only of XML-based customizations and JavaScript assets, are also packaged and deployed in the solutions gallery with a custom code assembly by default. If you are unable to access the source code to repackage any such solutions, they can no longer be deployed to SharePoint Online either. Unfortunately as it stands right now, the vast majority of the custom code developed for earlier versions of SharePoint cannot be migrated directly to the cloud. Please keep this in the back of your mind as we discuss the problem scenario for this course, and consider the ramifications for all the farm and sandboxed solutions you may have deployed in your own on-premises environments.

Introduction of Problem Scenario

So you've been hearing the rumors. You knew in the back of your mind that it was going to happen eventually. Your organization is leaving its on-premises SharePoint farm behind and is moving to the cloud. Did anyone consult the SharePoint developers about this? Probably not. So you start doing your homework and it looks like the Add-in model may be the way to go for cloud-ready SharePoint development. Maybe you've already heard about the SharePoint Add-in or app model before, but every example you've ever seen has been small, standalone, and simple. How can these add-ins possibly be used to meet all of your customers' complex requirements that have taken you years to develop and maintain with server-side code? Will they even work at all? In this course, you will learn that the Add-in model is capable and powerful enough to solve the kinds of problems that you've solved with server-side code in the past. Keep in mind, however, that you won't necessarily be packaging and deploying actual add-ins that correspond with each customization you need to build or recreate. You may not ever even need to touch your organization's app catalog. Occasionally you will only need to leverage techniques associated with the Add-in model to accomplish your objectives. You will see how JavaScript and the SharePoint client object model can be used to style and brand a SharePoint site, as well as replace the declarative XML-based customizations you've deployed in the past for site columns, content types, and custom actions. You will even learn how those custom timer jobs you built in the past can be deployed to the cloud, and made more flexible and reliable than ever before. Best of all, you can start taking advantage of these techniques right away, even before you move to the cloud, as they are also fully supported in SharePoint 2013 and 2016 on-premises. This can enable you to transition more gradually to the cloud and reduce risk, no matter when or if your organization chooses to make the leap to the cloud, or even if they choose to adopt a hybrid deployment of cloud and on-premise SharePoint. Everything we discuss and demonstrate in this course should be transparent to your end users in terms of where or how a particular customization is implemented using the Add-in model. Now we will examine the current on-premise SharePoint site that needs to be migrated to SharePoint Online. We will look at the types of customizations that have been made to the site using farm and sandboxed solutions that must now be redone in order to be cloud ready. Here is the home page of our on-premise SharePoint site. As you can see, besides being focused on the weather, several changes have been made to this site so that it does not look like a typical out-of-the-box SharePoint site. These changes include a custom master page, custom branding, and a custom web part that displays personalized data to the user. Some of these customizations were deployed via farm solutions, while others were deployed via sandboxed solutions. Together these make up our user interface customizations. If we navigate to our Site settings, we can examine the customizations that have been deployed to our Web Designer Galleries. For instance, under Site content types, we see a Custom Content Type, which if we drill down into its definition, contains a handful of custom site columns. Along with this we have defined a custom list instance that leverages this custom content type. All of these items were defined in XML files and deployed via sandboxed solutions in the Site Collection Solutions Gallery. These make up our declarative customizations. Now let's say for the purposes of re-use that I wanted to save this site as a template. In fact, I could do just that by going to Site settings and selecting Save site as template. That site template would take the form of a WSP file. A sandboxed solution in fact, that contains among other things all of the lists, libraries, content types, and master pages within my site. If I were to attempt to deploy this sandboxed solution in another environment, I would quickly learn that the solution contains hardcoded dependencies on features activated within this site that may have been installed by other farm solutions. This means this approach will not work in SharePoint Online. Finally we can check in Central Administration of our on-premises environment to see if there are any custom timer jobs defined. If we go under Monitoring, and then select Review job definitions, we in fact see that there is one custom timer job that is scheduled to run daily. This timer job was deployed via a farm solution, and activated via a web application scoped feature. This is our timer job customization, which also cannot be deployed to SharePoint Online. And here is a sneak peek of our brand new SharePoint Online site, running in Office 365. This is the site that must ultimately replace our current on-premises site. In the modules ahead we will transform this site to mirror the look and feel of our current on-premises SharePoint site, and reproduce all of the functionality you've seen in this demo using techniques from the SharePoint Add-in model. We will start by implementing our user interface customizations in module 2. Then we will implement our declarative customizations in module 3. Finally we will implement our timer job customization in module 4. Feel free to jump directly to any of these modules, or between them as you see fit. At this point you may be wondering, or desperately hoping, that there is a simple utility you can run to convert your existing farm and sandboxed solutions to SharePoint Add-ins. Unfortunately such a tool would be impossible to build because of the countless different ways that customizations can be implemented in SharePoint. For instance, some of the custom user interface elements you saw on the home page of our on-premise site could have been implemented in any different number of ways, via a custom web part, via markup embedded directly within the master page, or some other way entirely. Unfortunately a nontrivial amount of unavoidable rework may be necessary to recreate existing functionality using techniques from the Add-in model. This is work that must absolutely be undertaken before migrating to SharePoint Online, but is also recommended before upgrading on-premise SharePoint farms to SharePoint 2013 or 2016 as Microsoft continues to encourage developers to move away from farm and sandboxed solutions.

Taking Inventory of Your Custom Code Portfolio

As you saw in the previous demo, all of the customizations required to implement the functionality you saw in our on-premise SharePoint site were accomplished through farm and sandboxed solutions, but a number of questions may have crossed your mind while we were looking at everything. Questions like what solutions correspond with what customizations? How many solutions are there total? Do I really need all these solutions, or are there any solutions that may not be needed anymore? Finding these answers is critical as you assess the scope and magnitude of the development effort that will be required to migrate your legacy customizations to the Add-in model. As a seasoned developer, you likely already have a sense of this, but it always helps to be armed with specifics as you discuss the timelines for necessary rework with your management. Depending on the size of your SharePoint farm, you can employ a number of different techniques to get a feel for how much custom code is currently deployed in your environment. For larger farms you will want to leverage PowerShell and run scripts that can iterate through all the web applications and site collections in the farm, outputting a list of all deployed farm and sandboxed solutions. This list can then serve as the starting point for important discussions you will need to hold with other developers, management, stakeholders, and end users to determine what functionality is still needed, and what may be left behind. In smaller environments, or if you were only focused on migrating 1 or 2 sites at a time, you may find it easier to navigate through SharePoint's user interface to examine the Site Collection Solutions Gallery for sandboxed solutions, and the list of farm solutions in Central Administration. When using PowerShell, there are two cmdlets that will be most valuable to us as we take inventory of our custom solutions. The first, Get-SPUserSolution, takes a site collection URL as a parameter and will return a list of sandboxed solutions deployed to the Solutions Gallery, along with their solution IDs and whether or not each sandboxed solution is currently activated. The second, Get-SPSolution, will return a list of farm solutions, along with their solution IDs, and whether or not each farm solution is currently deployed. In this demo, we will take a more precise inventory of the customizations that have been made to the site using farm and sandboxed solutions that must now be redone in order to be cloud ready. We will also see how to examine the contents of a deployed WSP file to determine exactly what it contains. Since our problem scenario consists only of a single site that needs to be migrated, it is easy for us to look directly in the Solutions Gallery of this site to see how many sandboxed solutions are deployed here. As you may recall, we can get to the Solutions Gallery by going to Site settings, and then choosing Solutions under Web Designer Galleries. In this case we see two sandboxed solutions, one whose name suggests it contains our custom list definition with site columns, and one which appears to be a responsive CSS implementation. Now let's say it wasn't so simple to determine what functionality was packaged in a particular solution. In that case, we would need to do a bit more digging. I'll show you what I mean. From the solutions gallery I can click on one of these solutions and have the ability to download the actual WSP file for that solution. Once I do that, I can take advantage of the fact that a WSP file is nothing more than a cabinet archive file, or CAB file, and rename the file in Windows Explorer to have a .cab extension. After changing the file extension, I can double-click this file and open it. Right off the bat you'll notice this particular package contains a DLL file. As I mentioned before, even sandboxed solutions like this one that contain no custom code will still contain a custom code assembly by default. That means we cannot directly deploy this solution package to SharePoint Online unless we repackage it without this assembly. I'll show you how we can do that in module 3. Another way to view the list of deployed sandboxed solutions is with PowerShell. If I launch PowerShell and run the Get-SPUserSolution cmdlet, you'll see the same two sandboxed solutions we saw when looking in the Solutions Gallery. This approach doesn't give you the ability to directly download any of the solution files, but using PowerShell is far more scalable in larger environments, because I can first call Get-SPSite to get a list of all site collections in the farm, and then for each of those sites call Get-SPUserSolution to get the comprehensive list of sandboxed solutions deployed in all site collections in the farm. That cmdlet looks like this. In this environment since we only have one site collection, you'll see the same results in both instances. Here in Central Administration, I can see all of farm solutions that have been installed by going first to System Settings, and then to Manage farm solutions. Clicking any of the links you see here will not download the original WSP file like we could in the Site Collection Solutions Gallery for sandboxed solutions, but it will show us the deployment status of the farm solution, including whether or not it contains an assembly that's deployed to the global assembly cache, as well as when and where the solution was last deployed. If we do need to download the WSP file, perhaps to do the same kind of examination of what exactly is contained within a particular solution as we did before, we can do that using PowerShell as well. Here we are back in PowerShell. First I'll show you how to get the list of installed farm solutions. The cmdlet is simple, Get-SPSolution. Now let's say I want to crack open the poorly-named sharepointproject1.wsp solution so I can determine what it actually contains. To do that I'm going to run the following series of cmdlets. First we're going to get a reference to our SharePoint farm and store it in a variable. Then we will get a reference to this specific solution in the Farm Solutions collection. Finally I will call SaveAs on the file to save it to the local file system. Now I can navigate to this location and change the file's extension to .cab as you saw me do earlier. Once I've done this, I can double-click the file and examine its contents just as we did with our sandboxed solution. In this case the solution contains an application page. We can open the manifest.xml file to find out more details about where this page is deployed. I can do that by double-clicking it and extracting the file. Once that's done I can open it, and from here we will see this application page is deployed to the Layouts\SharePointProject1 folder. Hopefully you will find these techniques useful as you take inventory of your codebase customizations to SharePoint and determine what solutions correspond to what customizations.

Characteristics of Add-in Model Solutions

What is it exactly that makes an Add-in model solution an Add-in model solution? You will see some recurring themes as we update our legacy customizations throughout this course: the use of client-side code to include both plain old JavaScript and SharePoint's JavaScript client object model, remote provisioning of assets, a powerful and flexible way to deal with the manner in which files, as well as things like site columns and list definitions are initially deployed to and configured within a site, and custom code, when it is necessary, will only run on a server external to SharePoint, and will communicate with SharePoint via the managed .NET client object model or SharePoint's REST APIs. Now let's delve into each of these areas in a little more detail. Some add-ins you develop will be very simple and straightforward, consisting of some basic HTML and CSS, with maybe a few lines of JavaScript. When combine with jQuery, this is often all you need to make basic tweaks to SharePoint's user interface, through simple manipulation of the DOM. This is client-side scripting. In more advanced scenarios, you can leverage SharePoint's JavaScript client object model or JSON to access and manipulate data stored in SharePoint, or perform more advanced tasks, such as creating list items, libraries, or entire sites. Remember that SharePoint also offers a managed .NET client object model, or CSOM, which allows you to communicate with SharePoint from external clients or servers. Other solutions may require some custom server-side code that runs once, just to manage the initial deployment of assets across a site or series of sites. This technique is known as remote provisioning, and it enables you to use SharePoint's client object model to easily deploy customizations such as master pages, custom site columns, content types, and list instances to a site, without that messy, hard to maintain XML syntax you may have used in the past in conjunction with the Feature framework. Depending on your specific scenario, you could use the .NET managed client object model from an ASP.NET web application, or a simple console application running on a client workstation. Remote provisioning is an extremely powerful concept, and forms the basis for much of what can be considered modern SharePoint development, particularly when combined with the Add-in model. Remote provisioning is code-centric, which is great news for us developers. It is template-based, which means it is easy to understand, customize, and reuse. It also works both on-premises and in the cloud, so you should consider using this technique for any new SharePoint development efforts. We will leverage remote provisioning in module 2 when performing some of our user interface customizations, and then again in module 3 as we perform our declarative customizations. Of course there will always be scenarios where custom server-side code in a web application is still the only way to meet your customers' complex business requirements. These are situations in which there is truly no viable client-side alternative way to build something. However, the server-side code you will be writing no longer leverages SharePoint server object model, and it won't be packaged as a SharePoint solution package. That's right, no more SP web objects to dispose. Instead, your custom code will run somewhere outside the SharePoint server. This could be in Azure, AWS, or some other on-premises web server, but it won't ever be the SharePoint server. This paradigm shift is great for on-premises and hybrid environments, because it protects the SharePoint servers from the inherent risks of farm solutions, and custom code running in SharePoint's IIS worker processes. Of course it is also a must have in the cloud where custom code cannot run in SharePoint Online under any circumstances. You may have thought about this already, but in fact, this even means that you are free to develop code in a completely different language that runs on a completely different server platform, while still seamlessly communicating with SharePoint. This opens up the world of SharePoint development to any .NET developer who can leverage the CSOM, or to any web developer who knows how to interact with a REST API endpoint. This is a substantially larger number of developers than those who know the SharePoint server object model.

Introduction to Office Developer Patterns and Practices

Because the Add-in model represents such a strong departure from the way things were done in the past with full trust code in SharePoint, Microsoft started the Office Developer Patterns and Practices, or PnP initiative, to assist developers with transforming their existing solutions to be cloud ready as they migrate to SharePoint Online in Office 365. We will leverage the PnP PowerShell extensions, as well as what is known as the PnP Core component throughout this course. The PnP Core component contains a series of client object model extension methods for reducing the time and complexity required to code Add-in model solutions for SharePoint Online and on-premises. It also includes the Remote Provisioning engine or framework we will use in modules 2 and 3, and the remote timer job framework that we will discuss in module 4. The PnP Core component gives SharePoint developers new to the Add-in model the ability to build reliable solutions and implement common customizations using simple, fully supported building blocks without having to write lots of extra code. The PnP Team maintains several GitHub repositories that include reusable, production-ready components, templates, and solution starters. They also provide extensive samples, scenarios, solutions, and guidance to assist customers who are migrating to Office 365. In this demo, you will see how to leverage the PnP Core component in your Visual Studio projects. We will add the PnP Core NuGet package to our project, and illustrate the concept of remote provisioning by writing a console application that creates a list in SharePoint Online. Here we are inside Visual Studio 2015. In this demo I will show you the concept of remote provisioning, but without using the Remote Provisioning engine that you will see later in this course. I will begin by creating a new console application project. Then I will add the Office Developer PnP NuGet package to my project. I can do that by doing to Tools, NuGet Package Manager, Manage NuGet Packages for Solution. I will select Browse, and then I will search for officedevpnp. You'll see a number of results come back, some of which have been deprecated. When targeting SharePoint Online, you will want to select SharePointPnPCoreOnline. Notice that there are also options for targeting SharePoint 2013 and 2016 on-premises. I will install the latest stable version to my project by selecting it, selecting my Project, and clicking Install. This will add all the necessary assembly references and dependencies to my project, which are shown in this dialog. After I accept the various licenses, Visual Studio will finish configuring my project. Now that we have successfully added the PnP NuGet package to our project, let's write some code. For those of you who have written code using the .NET managed client object model before, you know that it can take dozens of lines of code to perform relatively simple tasks, such as creating a new list. I've added some using statements to my console application code, including most importantly Microsoft.SharePoint.Client. Keep in mind the PnP Core component exists as a series of extension methods that are implemented in that namespace. Now I'm going to paste some code here, and you'll see that the PnP extension method to creating new list is just a single line of code. This application is going to create a list called My Demo List in my SharePoint Online site. Let's go ahead and run that application right now. You'll notice it says the list has been successfully created, so if I hop over to our SharePoint Online site, and I refresh the Site Contents screen, you'll see that My Demo List has been successfully created in this site. We've only begun to scratch the surface of what can be accomplished using the Office Developer PnP tools. Stay tuned in the next modules as we begin to implement our legacy customizations using Add-in model techniques.

Moving User Interface Customizations to the Add-in Model

Introduction

Hi, this is Danny Jessee, and I would like to welcome you to module 2 of this course, Moving User Interface Customizations to the Add-in Model. In this module, we will describe the ways that front-end user interface customizations deployed via farm or sandboxed solutions can also be performed using Add-in model techniques. The benefits of using the Add-in model are undeniable. By moving custom server-side code outside of SharePoint, and leveraging JavaScript and SharePoint's client-side object models wherever possible, SharePoint runs with a much greater degree of reliability. You will see how remote provisioning can be used to deploy things like custom master pages, page layouts, and branding assets without requiring any file system footprint. You will also learn how to avoid the need for custom master pages in the first place, by making all of your front-end customizations via JavaScript, jQuery, and custom CSS that can also be deployed and configured without using farm or sandboxed solutions. You will see how composed looks and themes can be used as a way to implement simple branding changes for team sites. Finally, we will see how custom web parts can be moved to the Add-in model through the use of app parts and app script parts.

Master Pages

Let's start by talking a little bit about custom master pages. When SharePoint first adopted the master page model beginning with SharePoint 2007, organizations took advantage of the ability to brand their SharePoint sites by creating custom master pages. These master pages were deployed to the Master Page Gallery using farm solutions, and everyone was happy. Well, at least everyone on the Corporate Identity and Marketing Teams. All was well in the world, until it was time to upgrade to SharePoint 2010. For the vast majority of unprepared organizations, this major version upgrade caused major pain. The new default look and feel of SharePoint, represented by v4.master, completely wiped out all the custom changes that had been implemented. Significant amounts of rework were necessary, as Microsoft had introduced new controls and changed the names of important content controls and placeholders. Nonetheless, many organizations continued to implement and deploy custom master pages to achieve a desired look and feel, and the advent of sandboxed solutions made it even easier to deploy custom master pages scoped to a particular site collection. With SharePoint 2013 came yet another new look and feel, represented by new master pages, seattle.master and oslo.master. What's more, SharePoint 2013 also introduced the Design Manager, which made it significantly easier for web designers who were unfamiliar with ASP.NET or the master page model to develop master pages and page layouts using HTML and simple placeholder snippets that SharePoint could transform into a custom master page. By eliminating this technology barrier, even more organizations got in on the act of creating and deploying custom branding for their SharePoint sites. What happened when it was time for these organizations to migrate to the cloud? By this time, most of them had already figured out how to deploy custom branding via a sandboxed solution, so deploying that to SharePoint Online wasn't a problem, at least not in the early days of SharePoint Online. So all was well again, right? Well, it was until Microsoft rolled out the new suite navigation and app launcher. See that waffle icon in the top left corner of the screen? If you were using a custom master page, your users would not see it, or any of the valuable links it contains. What's worse, if you overwrote the out-of-the-box master page in SharePoint Online, your master page had now become un-ghosted, and would no longer have any updates applied to it, no matter when or how Microsoft rolled them out. Because of situations like this, and due to the frequency with which Microsoft intends to push updates to SharePoint Online, Microsoft's official recommendation is that you not utilize custom master pages at all. This about face from Microsoft with respect to custom master pages has likely left you a bit shell shocked. You're probably watching this course because your organization has made a significant investment in custom branding for SharePoint. Are you doomed to be at the mercy of Microsoft with the default look and feel of SharePoint Online moving forward? Not at all. In fact, you don't have to change anything with your current custom master page if you don't want to. Just keep in mind that going this route may require a degree of vigilance on your part. For instance, if Microsoft pushes important new functionality to the default master page, such as the app launcher we just saw, you would need to manually track down these changes, determine if they need to be included in your custom master page, and if so, implement these updates yourself. There are also several content placeholders that Microsoft expects to be on all master pages that will cause errors if they do not exist, and will prevent users from being able to access your site. The preferred option going forward is to make all of your user interface customizations with a lighter touch. In other words, avoid customizing the default master page whenever possible. JavaScript, jQuery, and custom CSS make nearly any front-end modification just a matter of defining the proper selectors and styles with a little bit of HTML thrown in. You'll see what I mean when we discuss branding in more detail, later in this module. You've probably heard about page layouts in addition to master pages. If your organization makes use of publishing sites, you may have even developed a few custom page layouts using SharePoint Designer, or the Design Manager in SharePoint 2013. A page layout is a template for a SharePoint publishing page that defines the presentation of content on that page. It is generally tied to a specific content type with certain page fields. Organizations that make extensive use of SharePoint's publishing features often leverage custom master pages together with custom page layouts to define a complete brand identity with precise positioning of page elements. Like master pages, page layouts are also deployed to a site's Master Page Gallery. When we demonstrate the deployment of custom master pages using remote provisioning, you will see that your custom page layouts can be handled in exactly the same way. In this demo, we will remotely provision a custom master page and other branding artifacts to SharePoint Online, using the PnP Core component we introduced in module 1. Although Microsoft no longer recommends the use of custom master pages in SharePoint Online, using remote provisioning in this case can serve as an important first step toward reducing your overall dependency on sandboxed solutions as you transition to the cloud. If you've made extensive changes to implement custom master pages and page layouts in your publishing sites, you may prefer to continue using these assets as you move forward with your migration to the cloud. As a reminder, we are now looking at our current on-premises farm. This site makes use of a custom master page that was deployed via a farm solution. In this demo, we will leverage a sample application provided by the PnP Team that uses the Core component to remotely provision this master page file and other images to my SharePoint Online site. It will then set my site's master page to use this file. Keep in mind, this isn't what I've been referring to as the preferred modern approach to SharePoint development, because we are still using a custom master page with a lot of embedded markup, but it is a step in the right direction, and it may be the only step you need to take if your organization makes the conscious decision to continue using custom master pages. Remember, custom master pages are still supported, just not recommended if you can avoid it. Now I could crack open Visual Studio and develop a new console application from scratch that leverages the core component to remotely provision these assets to SharePoint Online. You saw me do something similar in module 1 when I used the core component to create a new list in my site. However, in many cases you will find that the Office Developer PnP Team has already developed a sample application that may meet your needs entirely, or will at least help you get started, without requiring you to write much additional code. Many of these applications are configuration file driven, and will not require you to write any code at all. The easiest way for us to get started is to just download the entire PnP GitHub repository. If I go to github.com/OfficeDev/PnP, I can download a ZIP file of everything in the repository by selecting Clone or download, and then Download ZIP. I'll go ahead and save that file, and once it finishes downloading, I can extract the contents of this file and browse what's inside. I'll start by looking under PnP-master and then Samples. As you can see, there are tons of sample applications here. The PnP Team has really provided us with a wealth of resources to assist with many common tasks you may encounter while migrating your legacy customizations to the Add-in model. In this case we're interested in Branding.ApplyBranding, and now I'll open the solution. And once that opens you will see the solution contains a readme file, which I encourage you to review. You will also notice, as I mentioned before, many of the PnP solutions are XML configuration file driven, and this one is no different. Under the Branding.ApplyBranding project you will see a settings.xml file, and if I open that you will see this XML file contains a series of elements that we can customize. The branding element, which is the root element of this file, contains the URL to our SharePoint Online site collection where we will be remote provisioning these assets. The sites element is where we will specify a list of one or more sites where these assets will be remote provisioned. The files element is where we will specify a list of files to remote provision, along with the path, such as the style library where these assets will be stored. The masterpages element will contain a list of one or more master pages, which will of course be deployed to the Master Page Gallery, as will any page layouts we specify under the pagelayouts element. Now you may notice within this XML file that we aren't specifying any absolute paths to the files we'll be remote provisioning. That's because this solution assumes that all of those files will exist under the Branding folder within the project. If I expand this out you'll see that there are folders for files, master pages, and page layouts, and these correspond to the items being referenced in the settings.xml file. I'll go ahead now and make updates to the contents of these folders and the configuration file to meet the requirements of our on-premises site. I've gone ahead and grabbed a couple of assets from our on-premise site to include the logo image and the background image, as well as the custom master page file. There's one additional change I need to make before I run this, and that's in the project properties. I need to specify the default behavior in the command-line arguments. In this case we are activating the custom branding solution, and we're doing it in SharePoint Online. So I've added the command-line arguments activate online. I will now go ahead and run this solution, it's going to prompt me for my SharePoint Online username and password. Now it's going to begin provisioning these files to my SharePoint Online site, as well as upload and apply the master page. And it's done. If we come to our SharePoint Online site and refresh the Site Contents screen, you will notice that our custom master page has been applied, complete with its header div and background image. You'll also see some images have been deployed to the Images library, however, you might notice a few things don't look quite right either. Our site logo image doesn't appear, and the top navigation section doesn't look like it did before. Most notably the app launcher or waffle that we normally see in our SharePoint Online site does not appear in the top left corner of the screen. Those are some of the potential risks of taking a custom master page from SharePoint on-premises to SharePoint Online. Coming up later in this module, I'll show you a better way to make these changes without having to deploy a custom master page.

Branding

Let's talk a little more about branding. When we refer to branding in the context of SharePoint, we are talking about the intentional, coordinated use of visual elements, such as colors, fonts, and images such as logos that allow the look and feel of a site to match an organization's brand or identity. Many organizations have made tremendous investments in developing their brand, and oftentimes corporate style guidelines have ended up on the SharePoint developer's desk with a mandate to make SharePoint reflect the organization's branding. This occasionally results in solutions that may look great, but may not be the most maintainable or cloud ready. As you saw in our example on-premises site, one option for deploying custom branding to SharePoint Online is simply to embed lots of markup in custom master pages and page layouts, upload them to the Master Page Gallery, and call it a day. This may be the easiest approach for a web developer who is accustomed to embedding styles and hard coding changes to user interface elements directly within a page. While this gets the job done without using farm or sandboxed solutions, it misses the point when it comes to the preferred modern approach to making front-end customizations. As a SharePoint developer, designer or brander, your goal should be to keep your customizations as loosely coupled to SharePoint as possible. In the case of custom branding, this means externalizing your changes using JavaScript and jQuery to update the DOM, and CSS to make style updates whenever possible. This may be easier to accomplish for a team site than a publishing site, but you should strive to use these techniques on every site. I'll show you what I mean as we move through this module. Let's start by talking about custom CSS. In the past you may have needed to override one of SharePoint's globally defined styles to make elements on a page look a certain way. Maybe you needed every web part title on the page to appear in a larger font, or a different color. You used your browser's built-in developer tools to identify the proper CSS selectors for the elements you needed to change, and you come up with your new style definitions. One way these changes could be applied globally in your on-premises environment is simply to modify one of SharePoint's core CSS files directly on the file system. This would get the job done, at least in the short term, but you should never overwrite these default files. In much the same way that new versions of SharePoint introduced new default master pages that can render many of your previous customizations unusable, they also include new default style sheets that could replace or overwrite any changes you had made to the out-of-the-box style sheets. These kinds of changes could even be introduced in a service pack or cumulative update, and impact you when you least expect it. The recommended approach for making changes to SharePoint styles is to externalize them in their own CSS file. This notion of externalizing changes, which is the basis of the remote provisioning paradigm, will continue to be a recurring theme throughout this course, not only when we discuss branding, but all other types of SharePoint customizations as well. Now obviously in the context of the SharePoint Online there are no CSS files on the server file system that we will be able to directly modify, but even if you are staying on-premises, you should attempt to consolidate all of the style changes you need to make within a single CSS file, remote provision that file to your site, and then leverage something called the site alternate CSS property to reference that file. This property can be set through the SharePoint user interface, or through code using the client object model. Most of us have had to make simple tweaks to a page using JavaScript at one time or another. If we knew a particular element's ID on the page, we could find it and get a reference to it by calling document.getElementById, then we could make updates to that element. For example, we could change its color by saying x.style.color = blue. JQuery with its powerful syntax that leverages CSS selectors makes it even simpler to tweak things like an element's style or positon on the page, but keep in mind that we can use jQuery to do more than just update existing elements in the DOM. We can leverage our ability to manipulate the DOM to define brand new elements on a page, such as new div elements for things like page headers and footers. By properly traversing the DOM and identifying where our custom elements need to appear within a page's markup relative to existing elements, we can leverage jQuery in conjunction with a technique known as JavaScript embedding to ensure our script to modify the UI is executed on every SharePoint page. I will use this technique to add a custom header to all pages in our SharePoint Online site without having to modify the master page. JQuery can also be used to make powerful transformations to things like out-of-the-box list views to provide capabilities that you may have thought could only be done with SharePoint Designer or custom web parts. You will see some examples of that later in this module as we look at Add-in model techniques for replacing web parts. In this demo, we will examine the different ways in which custom branding was applied to our example on-premise site using markup embedded within the custom master page. In each case, we will look at a different approach that uses a lighter touch with client-side code and CSS. We will use remote provisioning techniques to deploy branding assets that may have previously been deployed using modules or mapped folders in legacy solutions, and we will use JavaScript embedding to ensure the scripts we develop to modify the SharePoint user interface are executed on every page. As a reminder, here is the look and feel of our current on-premises site. The customizations on the master page used by this site include some CSS tweaks to accommodate our larger than standard site logo by moving down the div containing the Quick Launch navigation. We also hide the breadcrumb navigation and search box. There's a hardcoded reference to a background image for the S4 workspace div, which is the main content area for the site. Finally, you'll see this header area at the top of the page, which is used for special notifications. Here is the markup for our current master page. Like many custom master pages out there, this one started by taking a copy of an existing master page, in this case seattle.master, and embedding changes within the page's markup. This particular master page was deployed via a farm solution, which you can see also deploys to images, the site logo, and background image to a subdirectory of the Mapped Images folder in the Layouts directory on the file system. Remember in our last demo when we remotely provisioned this master page in its current state to our SharePoint Online site, the results weren't quite what we wanted. The SharePoint Online app launcher was missing, and the top navigation area didn't look quite the way we wanted it to either. We will work around these issues and implement a cleaner, more maintainable overall solution by not replacing the default SharePoint Online master page, and instead applying our user interface customizations with a lighter touch. If we scroll down here, you will see where we have added our CSS changes. Instead of embedding these within the master page, we will externalize them in a separate CSS file. We will remotely deploy this file and set the site's alternate CSS URL property to match the deployed location of this file. This will result in the same CSS being applied to our site, but decouples our styles from the master page and makes it much easier for us to make updates to these styles down the road. Scrolling down further you will see the markup for our custom header div. We will write some JavaScript that leverages jQuery to dynamically insert this div in the DOM, without having to embed the change within the master page. We will then use a technique known as JavaScript embedding to ensure our script is loaded on every page within the site, giving us the same end result we currently have. The banner will still appear at the top of every page, but again, we will have accomplished this change without having to modify the default master page. To do this, I will need to identify where in the DOM I need to insert the markup for my div. Looking at our SharePoint Online site, I will use my browser's built-in developer tools to inspect this element, and I see that the ID of this particular div along the top of the page is suiteBarTop. JQuery provides a method called insertBefore that lets us specify some markup that will be inserted into the DOM before any elements that match a particular selector, so I'll go ahead and take the markup for our header div from the master page, and paste it in here, then call insertBefore the element with ID suiteBarTop. We'll wrap that with a jQuery document ready, and that one line of JavaScript is all we will need here. I'm going to leverage two sample add-ins from the Office Developer PnP Team in this demo. Remember in module 1 we downloaded the entire PnP GitHub repository to our developer workstation. All of the samples we're using can be found in the Samples folder. The first, called Branding.AlternateCssAndSiteLogo, is a provider-hosted add-in that uses the client object model to remote provision a CSS file and site logo image, then sets the site logo URL and alternate CSS URL properties of the site to reference these newly-provisioned files. Provider-hosted add-ins allow you to run server-side code on an external server, so you will see that the sample application uses the .NET managed client object model running within an ASP.NET web application. Best of all, for add-ins like this, which only need to be run a single time to provision the necessary assets and set the relevant properties of the site, we can deploy this add-in from Visual Studio directly to our SharePoint Online site. The server-side code that runs in the remote web can be executed directly from our developer workstation using IIS Express. The meat of this add-in lies within two functions in our remote web's default.aspx code behind, the button click handler that launches the process, and a function called UploadAssetsToHostWeb, which loads our site logo and CSS files to the Site Assets library of our SharePoint Online site. After the necessary files are remote provisioned, this function sets the alternate CSS URL and site logo URL properties of the site before updating the web. Just like the apply branding solution from our last demo, I will need to update this project to include my site's logo and CSS files under the Resources directory of the project. I'm also going to go ahead and include our background image and the JavaScript file that injects our header div into the DOM, since we will need to remote provision those files as well. I will also need to update the code to reflect the names of all the files that we need to remote provision. This code may not look ideal, but it does only need to be run once, and has already been written for us and blessed by Microsoft, so we shouldn't stress about it too much. I'll go ahead and make those updates to the project now. In order to connect this project to our SharePoint Online site for deployment, I have to enter my SharePoint Online site's URL in the Site URL field of the add-in project's properties. You may be prompted to sign in to your site. Now we also need to make sure we've selected the add-in project as our startup project, and now I can simply press F5, which will do 2 things. One, it will deploy the add-in from the add-in project to SharePoint. Two, it will deploy the ASP.NET web application project to IIS Express on my local machine. Now you'll see we have a deployment error. Let's not continue and see what's going on. You'll see the error says Sideloading of apps is not enabled on this site. Unless your SharePoint Online site happens to be a developer site, you too will receive this error when you try to deploy an add-in directly from Visual Studio to SharePoint Online. This is by-design behavior to prevent developers from bypassing the normal controls in place, to prevent you from loading add-ins directly to a site, but it can be overridden by a tenant administrator. Microsoft provides a PowerShell script to enable and disable sideloading that you can download from MSDN. Make sure you have the SharePoint Online management shell installed first. The recommended approach is to enable sideloading, deploy your necessary add-ins to SharePoint, and then immediately disable it again. I'll go ahead and run the PowerShell script to enable sideloading. Now that sideloading is enabled, I'll try my F5 deployment again. I'll press the Trust It button and you'll see we get redirected to localhost, which is the IIS Express instance running on our developer workstation. The code we updated to reflect the files we are using gets executed when we press the Run scenario button, so I'll go ahead and do that now. Remember, this provisions the files to our host web's Site Assets library, and sets the logo image and alternate CSS URL to point to the proper files. I see a message stating that the custom CSS and logo have been applied, so I'll go ahead and click this link, which will take us back to our SharePoint Online site. You'll see the updated styles have now been applied, rendering our site logo image at full size, pushing the Quick Launch links down to accommodate the larger logo. You can also see our background image. All that's missing now is our custom header div. I'm going to leverage another sample add-in from the Office Developer PnP Team to accomplish adding this custom header script to every page in our SharePoint Online site, thus ensuring our header div is rendered everywhere. This particular sample is called Core.EmbedJavaScript. This is also a provider-hosted add-in that follows a very similar paradigm to the alternate CSS and site logo add-in we just used. The meat of this add-in lies within two functions in our remote web's default.aspx code behind, the button click handler that launches the process, and a function called AddJsLink, which creates a script linked custom action that ensures a reference to the specified JavaScript file is included on every page. The default implementation of AddJsLink allows you to reference a JavaScript file that is permanently deployed to the remote web, but I can change the values of scenarioUrl and jsLink to point to the file we provisioned to our Site Assets library. I'm also going to add a second custom action here to reference jQuery from a content delivery network on every page as well. Our solution leverages jQuery, and other solutions are likely to make use of it too. This technique will ensure jQuery is also loaded automatically on every page. I've gone ahead and made all the necessary updates to this project, so now I'll press F5 to deploy this add-in. Much like the previous add-in we used, we'll press the Trust It button, and we are redirected to a site running on localhost. Again, that's our IIS Express instance on our developer workstation. We'll press the Embed customization button, and unfortunately this one doesn't give us any particular feedback when it either succeeds or fails, but once the postback returns I can click the link to go back to my site, and you'll see my custom header div now appears at the top of the page. That means that this page, as well as every other page within the site, now has jQuery embedded, as well as our custom header script to render this div. Our SharePoint Online site now has the same look and feel as our on-premises site, all with 0 farm or sandboxed solutions, 0 customizations to the master page, and 0 file system footprint. And best of all, our uncustomized default master page will continue to receive updates pushed down by Microsoft without any issues or concerns in the future. Now you see why applying a lighter touch using add-in model techniques such as remote provisioning and externalizing changes using CSS and JavaScript is the preferred approach when it comes to making user interface customizations to SharePoint, whether on-premises or in the cloud. Perhaps your organization doesn't have strict branding guidelines, or maybe you just don't have the time or energy required to create the assets necessary to make some of the customizations you've seen in this module. You still have the ability to update your SharePoint sites to use a color scheme, font scheme, and background image of your choosing, using what are known as themes and composed looks. Themes and composed looks are particularly useful on team sites, which usually don't require the same degree of extensive custom branding you may find on publishing sites. Both provide a quick and easy mechanism to freshen up a site with a new look using a custom color scheme, while composed looks also allow you to define a font scheme and background image for a site. Microsoft even provides a free SharePoint color palette tool to help you generate the color scheme file needed to define either a theme or a composed look.

Web Parts

Web parts are the most popular and most fundamental way to customize the SharePoint user interface. In this course, our focus will be on customizations developed as web parts deployed via farm or sandboxed solutions, not out-of-the-box web parts. SharePoint developers who leverage the Office Developer Tools for Visual Studio could quickly and easily create visual web parts, which provide a simple drag and drop approach to building a web parts UI using ASP.NET user controls when compared to previous approaches for building custom web parts. Originally, visual web parts had to be packaged in farm solutions, because they deployed user controls to the file system of the SharePoint server. Eventually support was added to deploy visual web parts in sandboxed solutions, but even those can no longer run in SharePoint Online because of the need for a custom code assembly. We will discuss add-in model alternatives to custom web parts, whether visual web parts or otherwise. That, like all the other web parts we know and love, can be deployed to a site's Web Part Gallery and added to pages by site owners and end users. The add-in model provides two approaches for replacing legacy custom web parts, app parts and app script parts. App parts can be deployed in SharePoint-hosted add-ins, which consist only of client-side script, CSS, and HTML that are deployed to an isolated add-in web on the SharePoint server. These are useful in simple use cases where server-side code is not necessary. App parts can also be deployed in provider-hosted add-ins, which do allow for server-side code, as long as it runs external to the SharePoint server. In either case, app parts load externally-hosted content by embedding an iframe on the page and loading the external content within that iframe. As you would expect, Microsoft has preserved the end-user experience of adding app parts to a SharePoint page from the Web Part gallery, just like custom web parts. Now you heard me mention iframes. Many of us may cringe when we think on iframes based on previous experiences we have had as web developers, but Microsoft has actually done a very good job of mitigating issues like nested scroll bars that caused us headaches with iframes in the past. App parts actually look as seamless within the page as traditional web parts, even though the content they service is being loaded from another domain. There is one unavoidable drawback to the use of iframes in app parts thought, they do not work well within a responsive design where the display of content on a page is scaled or altered based on the screen size or resolution of the device where the page is being viewed. There are some techniques you can use to help make iframes more responsive, but these require some additional tweaking using JavaScript and CSS, and are far from a complete responsive solution. With that in mind, let's talk a little more about app script parts. App script parts take their cues from modern web development paradigms that allow you to do things like embed a map or an external feed on web page by writing a few simple lines of HTML and JavaScript. The app script part generates HTML that is then added to the DOM within a specified container element, usually an empty div. Like app parts, app script parts load content from an external source, and can be added to a SharePoint page from a site's Web Part Gallery. Unlike app parts, however, app script parts do not make use of iframes. This allows you to more readily integrate app script parts into a responsive design. In this demo, we will recreate the functionality in the custom web part from our on-premises site using app parts and app script parts. We will also see how JavaScript and jQuery can be used to customize an out-of-the-box list view web part to avoid the need for custom web parts entirely. As a reminder, here is the custom web part from our on-premises site. It renders data from items in a custom lists called WeatherList. For the purposes of this demo, I've already gone ahead and created a copy of this list in our SharePoint Online site. I've also copied the image files you see here to our Site Assets directory. You've already seen how we could remote provision image files, and these are no exception. In module 3, we will see how to remote provision the site columns, content type, and list definition behind the WeatherList. The web part uses the server object model to query for items in that list, and renders the data by generating HTML markup for a table within an ASP.NET literal control, including an image for each item from a series of images that were deployed to a subfolder of the Mapped Images directory in a farm solution. For all of these reasons, this web part is not ready to be deployed to SharePoint Online in its current state. However, with a few simple tweaks, you'll see how we can recreate the functionality of this web part using SharePoint's JavaScript client object model to make either an app part or an app script part that is cloud ready. I'm going to start by creating an app part that is part of a new add-in using Visual Studio 2015. I'm using the free Visual Studio 2015 Community Edition with the Office Developer Tools installed. You can download the Office Developer Tools for Visual Studio 2015 from this URL. I will select New Project, and then SharePoint Add-in from the Office/SharePoint group. I will deploy this add-in to my SharePoint Online site, and since we don't need to write any server-side code, I can create this as a SharePoint hosted add-in. We will target SharePoint Online in this project, but notice you can configure your project to support SharePoint 2013 on-premises as well. Of course SharePoint Online may expose new APIs as Microsoft pushes updates over time, so that's why you have this option here if you don't need to worry about backward compatibility with SharePoint on-premises and always want to have access to the latest and greatest APIs. I'll press Finish and Visual Studio will create my project for me. To create a new app part, I can right-click the project and select Add, New Item, then choose Client Web Part from the list. Remember that the app part is an iframe that renders content from an external source, so I'm given the option of creating a new page or entering the URL of an existing one. I'll go ahead and create a new page for this demo, and press Finish. In the page that gets created, you'll see a reference to a WebPartPages:AllowFraming control, which is what tells SharePoint that it can render this page's content in an iframe. From there I'm free to make any additions or updates to this page I would like. I can add JavaScript and CSS references, as well as any custom HTML I may need. I'll go ahead and add my script to this page. I'm using the JavaScript client object model to query a list in the host web, which is the SharePoint site where our add-in will be installed. In order to be able to do this, I need to make use of SharePoint's Cross Domain library, which allows me to make JavaScript calls from the add-in to the host web. This is as simple as adding a script reference to sp.requestexecutor.js, which I've done here along with the other script references. As we scroll down, we'll see the implementation of this app part. We have a jQuery document ready handler that fires once the DOM is loaded. It calls retrieveListItems. RetrieveListItems will then get a reference to the app web and host web URLs, both of which are passed in via the query string. We generate a clientContext object, and most importantly, because we're making use of the cross-domain library, we're going to set the webRequestExecutorFactory based on the ProxyWebRequestExecutorFactory we instantiate from our app web's URL. From there we can create a new AppContextSite, passing in the clientContext and the URL to the host web. From there we're able to get a reference to the WeatherList, query it, and iterate through its items. In the onQuerySucceeded handler, we iterate through each of these list items and render some HTML markup based on the information in each item. Once this is done, we use jQuery to set the HTML content of a div with ID forecastContent based on this markup. Scrolling down the page, you'll see where I've added this div. I also need to make sure my add-in requests permission to read data from the host web. If I open AppManifest.xml and select the Permissions tab, I can have my add-in request permissions at the List scope; in this case we'll be requesting the Read permission. When I deploy this add-in, I will be asked to specify which list the add-in is allowed to read from. I'll now press F5 to deploy this add-in. Here I will select the WeatherList as the list that my add-in is trusted to read from, and I'll press Trust It. Now I am redirected to my Add-in start page. Microsoft is kind enough to provide a default implementation of this start page that displays your username. In production you could make this page into a readme file about your add-in. This is the page that will be displayed if the user clicks your add-in from the Site Contents screen, but for add-ins containing app parts, it's not something your users will likely ever interact with. Let me click this link at the top of the page to return to our host web. From here I can now add my app part to the page. You'll notice I've cleaned up some of the default web parts that were on this original team site to make room. I will edit the page, and then I will select Insert, and you'll see I have two categories here, App Part and Web Part. If I select App Part, you will see my App Part displayed here. If I select Web Part, the app parts are just displayed as a separate category, so either option works. Now that our app part is on the page we immediately see a couple of issues. One, because the page is being loaded in an iframe, it doesn't inherit the background image of our site. Two, the default dimensions of the iframe do not accommodate the content that is being displayed. Because our app part is 100% client-side code, it is tailor-made to be implemented as an app script part, which will mitigate all the issues we have just seen. I will use another PnP sample solution as a starting point to implement this as an app script part. The solution is called Core.AppScriptPart. The idea here is to take a .webpart file, which is something you may be familiar with from legacy web parts you may have implemented, and embed an external JavaScript reference and container div element within its contents. We will then remote provision this .webpart file to a site's Web Part Gallery, and when it is added to the page, the markup and script will run inline, rendering the same content you saw before, but within the same DOM as the page because it is not being loaded in an iframe. I've updated the PnP sample solution to include JavaScript that is nearly identical to what I used in the last example, but now externalized into a separate .js file. The sharePointReady function is called using the SharePoint script on demand framework to ensure that sp.js has been loaded, then does the same thing as the document ready handler did in our app part, using the JavaScript client object model to query for items in the WeatherList list and generating HTML markup in the onQuerySucceeded handler. The code that runs when the add-in is deployed will remote provision this file to our Site Assets directory. I've also updated the .webpart file to include a reference to this .js file, and to include the same div with id forecastContent that you saw me use before. This markup is encoded within the content property of the .webpart file, and represents what will be embedded within the page when this app script part gets added. I'll press F5 to deploy this add-in. As with the other PnP samples we have seen, this is a provider-hosted add-in that will launch the remote web and IIS Express on localhost. I'll select Trust It, and then the add-in's start page will load. When I press the Run Scenario button, the add-in will remote provision the .webpart file and my custom JavaScript file to the host web. Once that completes, I can return to the host web and add my app script part to the page. This time, however, our app script part will actually appear in the list of web parts under the Add-in Script Part group, which is what we specified in the XML markup of the .webpart file. Once we add this to the page, you will see a much better looking result. In fact, this app script part has generated output that looks identical to our on-premises web part that was deployed via a farm solution. Even using add-in model techniques, you see that a fair amount of effort can be required to generate the proper markup and deploy the necessary assets to make an app script part work. I wanted to show you one more option, which is great when an out-of-the-box SharePoint list view looks good, but you would like to make some minor styling tweaks to the output. On this page we have added a list view web part for our WeatherList list. Let's say I like the way this looked, but I wanted to render the Day titles in a larger font, and I wanted to display an image above the Conditions text. With the absence of Design view in SharePoint Designer 2013, it is no longer possible to make these types of user interface customizations using SharePoint Designer as you may have done in the past. JQuery still makes it possible to do all of this, however, and all I need to do to transform this list view web part is embed some JavaScript somewhere on the page, even within a script editor web part will do. Here is the script I have written, it looks for text that starts with Day and a space, and table cells with class ms-vb2, which is the CSS class applied to table cells in list view web parts. It then surrounds that text with h2 tags, which will increase its font size. It also looks for text and table cells with that same CSS class that match any of our weather condition strings, and updates the HTML of those cells to render an image file matching the condition name above the condition text. If I go back to edit my page and select Embed Code from the Insert tab, I can paste in my script. Once I insert this script and save the page, the script will execute and you will see the transformations have been made. So keep this in mind as another, even more lighter touch option as you look to make user interface customizations involving web parts.

Review

In this module, you have seen the value of remote provisioning branding assets such as master pages, page layouts, logos, and other images. You should always favor remote provisioning over the use of mapped folders or any other technique that would otherwise require a farm or sandboxed solution when you are simply adding files to SharePoint. Besides making it simpler to make minor changes down the road, without requiring any solution deployments or server downtime, this approach of externalizing all customizations with CSS and JavaScript more readily lends itself to making changes with a lighter touch, which is the crux of the preferred modern approach to SharePoint development. You have also seen your options when implementing custom web parts using the Add-in model, and the pros and cons of each approach. Throughout this module we took advantage of sample add-ins provided by the Office Developer PnP Team to give us a head start when performing tasks, like applying custom branding and styles, embedding JavaScript on every page within our site, and developing and deploying an app script part. As you saw by the end of this module, we had updated our SharePoint Online site to have the same look and feel as our on-premises site, without requiring a single solution to be deployed. In the next two modules, you will see how we can also implement more back-end types of customizations using add-in model techniques, leveraging assets from the Office Developer PnP Team where appropriate.

Moving Declarative Customizations to the Add-in Model

Introduction

Hi, this is Danny Jessee, and I would like to welcome you to module 3 of this course, Moving Declarative Customizations to the Add-in Model. In this module we will describe the ways that declarative customizations can be implemented using Add-in model techniques. When we talk about declarative customizations, we are referring to customizations that involve items that are defined using XML, such as site columns, content types, and list instances, rather than the front-end file-based customizations you saw in the last module. These types of customizations are frequently deployed as sandboxed solutions, although they can also be deployed as farm solutions. If you've been following along up to this point, you won't be surprised to hear that remote provisioning plays an important role in the preferred modern approach to handling declarative customizations. We will use the SharePoint client object model and the PnP Core component to remote provision declarative customizations to our SharePoint Online site. Up to this point we have leveraged simple, standalone solutions from the Office Developer PnP Team to handle the remote provisioning of assets such as master pages, JavaScript, and CSS files. In this module we will introduce the PnP Provisioning engine, which provides a more structured, holistic approach to remote provisioning everything from site columns, content types and list definitions, to composed looks, pages, and much more in a repeatable manner using provisioning templates.

Legacy Approaches to Declarative Customizations

SharePoint developers have long taken advantage of the built-in SharePoint project types provided by Visual Studio for developing SharePoint solution packages. Beginning with Visual Studio 2010, ensuring that the information architecture components needed by your SharePoint application were deployed as part of your solution was as simple as right-clicking your project and adding new project items using the built-in item templates for everything from site columns to content types and list instances, then associating these elements with Feature definitions. Early versions of these capabilities presented the developer with an XML file that needed to be manually edited to reflect the necessary attributes for each field element. Later versions made parts of this process less error prone by giving developers more friendly user interfaces and wizards to guide them through the process of adding these elements. In either case behind the scenes, each individual site column, content type, and list instance you create using Visual Studio project items generates an elements.xml file containing the definition of that specific item. When deployed via the Feature framework in a farm solution, these files are stored in subfolders beneath the feature's directory for your solution on the file system of the SharePoint server. In a sandboxed solution, although there is no file system footprint, these files are still deployed within a site's content database, and of course still leverage the same SharePoint Feature framework. As we learned in module 1, any sandboxed solution you build in Visual Studio will contain a custom code assembly by default, regardless of whether or not your solution actually requires one. We also learned in module 1 that sandboxed solutions containing a custom code assembly can no longer be deployed to SharePoint Online. Although deploying sandboxed solutions of any kind to SharePoint Online is no longer recommended, no-code sandboxed solutions without a custom code assembly are still supported. In the event you need to deploy a no-code sandboxed solution to SharePoint Online, ensure you have set the IncludeAssemblyInPackage property to false in your Visual Studio project before packaging and deploying the solution.

Remote Provisioning

The preferred modern approach to SharePoint development does not involve the use of the legacy SharePoint Feature framework, but instead relies on the concept of remote provisioning to deploy customizations to SharePoint with a lighter touch. Besides removing the need to deploy all of those elements.xml files, writing client object model code to manage site columns, content types, and list instances gives you total control over the entire deployment process. Rather than relying on the built-in tooling for creating and deploying declarative customizations, which can have the unintended consequence of doing undesirable things, like overwriting list instances when they already exist, you can implement any custom logic or business rules you may need, such as preserving data in an existing production list when all you need to do is add a new list column or make a small change to an existing one. It also allows you to make simple, incremental updates without requiring a full solution redeployment. You've seen me use the client object model several times already in this course, as we have remote provisioned files like master pages and branding assets. For the types of customizations we are focusing on in this module, I'd like to take a quick step back to review the paradigm we always follow when using the client object model, as it is a bit different from how you may have used the server object model in the past. First, we always begin by instantiating a ClientContext object based on the URL to the SharePoint site we will be working with. Then we define our query and load the objects and any associated properties or collections we need to access or manipulate, such as the web and its fields, lists or content types. Then we call ExecuteQuery to actually send the query to the server. This allows us to batch the requests we make to SharePoint, which can increase efficiency and reduce the overall load on the SharePoint server. However, it also requires you to be more diligent about ensuring you structure your queries properly, and always call load on any objects whose properties you need to access or manipulate, before attempting to do so. Code snippets like this probably look familiar to you if you have done SharePoint solution development in the past. This code creates a new site column and content type, then adds the new site column to that content type. It has always been possible to deploy declarative artifacts in this way, using server object model code instead of XML, and many developers opted to implement code like this in feature receivers that would execute whenever a particular feature was activated or deactivated. As we move away from leveraging the legacy Feature framework, however, code like this will need to be retooled to make use of SharePoint's client object model. Here is code that does the same thing you saw in the previous slide, but now implemented using the SharePoint managed .NET client object model instead. You'll see that it takes a few more lines of code to accomplish the same tasks, but we have now offloaded this processing from the SharePoint server. Notice all the calls to Load and ExecuteQuery that are now necessary, which we did not need to worry about when using the server object model. Finally here is the same client object model code from the previous slide, now implemented using APIs exposed by the PnP Core component. Again we see how the core component provides us with functions that drastically simplify common tasks, such as creating a field, creating a content type, and adding a field to a content type by name, while also reducing the number of lines of code required. In this case we have reduced the number of lines of code from what you saw in the previous slide by about 75%. Notice also the absence of explicit calls to Load and ExecuteQuery. That's because the APIs from the core component encapsulate those calls for us, so we don't have to worry about remembering to do that anymore either. In this demo, we will use the SharePoint client object model in conjunction with the PnP Core component to remote provision the site columns, content type, and list instance from our on-premises site to SharePoint Online. To review the information architecture from our on-premises site, we have custom site columns for Conditions, ForecastLastUpdated, HighTemperature, and LowTemperature. These site columns were then added to a new content type called Weather. Finally the WeatherContentType was added to a new list instance called WeatherList. These customizations were all deployed using the Feature framework in a sandboxed solution. You can see each customization is its own item within the feature, each containing its own elements.xml file. This is what permanently associates each of these customizations with the solution, and why serious problems can arise if the solution were to be retracted or redeployed. Of course the way to properly address these issues is to remote provision our artifacts using the client object model. To do this, I'm going to create a new console application in Visual Studio, and add the NuGet package for SharePointPnPCoreOnline, as you saw me do in module 1. As you will recall, this adds all the necessary references to my project for the SharePoint client object model, as well as the PnP Core component, which is going to help drastically reduce the number of lines of code I need to write to apply these changes. When I run this application it will connect to my SharePoint Online site, and remote provision the site columns, content type, and list instance using that content type. I'll go ahead and add my code. Now let's take a closer look at this code. First we create the four custom site columns using the web.CreateField method of the core component, then we call web.CreateContentType to create our custom content type, followed by four calls to web.AddFieldToContentTypeByName. Each of the GUIDs you see in those calls correspond to the GUIDs we used for the ID values of each field when we created them, using the web.CreateField calls above. Next we call web.CreateList to create our WeatherList. And finally, web.AddContentTypeToListByName to add the WeatherContentType to our WeatherList. Factoring out our console.write statements, we have accomplished in 11 lines of code what would have easily taken 40 or more lines without the PnP Core component. I'll go ahead and run this application. We see success across the board, so let's hop over to our SharePoint Online site and verify that all the customizations have been applied. We can see that our WeatherList appears here, and if I navigate to that list and look at its settings, we see our Weather content type has been added to the list, along with the custom site columns from that content type. We have successfully applied our declarative customizations using remote provisioning, with a little help from the PnP Core component.

PnP Provisioning Engine Introduction

So far we have seen numerous examples of how to perform incremental, piecemeal deployments of capabilities using Add-in model techniques. We deployed one add-in to apply our branding, then another to implement some of our declarative customizations. What if there was a way for us to quickly capture all of the relevant customizations within a site, then apply them in a new environment all at once, in an automated, repeatable manner? This is where the PnP Provisioning engine comes into play. You may have utilized SharePoint's save site as template functionality in the past, which generated a solution package that could then be uploaded to the Solutions Gallery, and activated on a target site to apply the customizations from the source site. The PnP Provisioning engine allows us to accomplish this using Add-in model techniques by remote provisioning required assets and using client object model code to capture and apply configuration changes to a site. It captures customizations from a source environment, such as its site columns, content types, list instances, permissions, and much more in a provisioning template file that can then be parsed and applied in a destination environment. Provisioning templates can also specify lists of files and folders that should be copied from a source site to a destination site, such as the contents of a Site Assets library to ensure branding changes are applied. Provisioning templates are particularly useful when you need to deploy a series of customizations to multiple environments. Perhaps you have separate development, staging, and preproduction environments in addition to your production environment. Using a provisioning template enables you to adopt a write once, apply everywhere approach that can reduce risk and ensure repeatability as one provisioning template can form the basis for any number of destination sites. The process of generating a provisioning template requires the use of PowerShell. To generate a provisioning template, you must first download the PnP Core PowerShell extensions from one of these URLs. You can install the extensions for SharePoint on-premises and SharePoint Online on the same machine. Once the extensions are installed, you can generate a provisioning template file by running the cmdlet Get-SPOProvisioningTemplate. This process captures the customizations made to the site, and generates the provisioning template file that we will specify when running the cmdlet to apply the provisioning template to the destination site, Apply-SPOProvisioningTemplate. You may be thinking that provisioning templates would be a good way to capture the customizations made in an on-premises environment to then apply directly in SharePoint Online. Depending on how your customizations were originally deployed to your on-premises environment, however, this is probably not the case. Unfortunately, anywhere you have deployed declarative artifacts via solutions and the legacy Feature framework, the IDs of any associated features are permanently registered as dependencies of that artifact, meaning those artifacts cannot ever be decoupled from the Feature framework. The provisioning engine will capture these dependencies when generating the template file. This is of course the exact kind of problem we are looking to solve going forward, by applying Add-in model techniques to our SharePoint customizations. The best way to work around this limitation is to create a new site in the target environment, then use the techniques you have already seen throughout this course to apply of your customizations without using solutions or the Feature framework. In the upcoming demo, we will use the same SharePoint Online site you have seen me use throughout this course to generate our provisioning template. Once you have created the template file and make any necessary adjustments to it, you can then apply it to any number of staging or production sites where you would like to deploy these customizations.

Using the PnP Provisioning Engine

In this demo we will use the PnP Provisioning engine to create a provisioning template based on our SharePoint Online site. We will customize this template by making some manual updates to it by hand, and then we will apply this provisioning template to another site collection in SharePoint Online. We will go ahead and start by launching the SharePoint Online Management Shell. To connect to our SharePoint Online site, we will run the cmdlet Connect-SPOnline. This will prompt me for a URL, and then the credentials to access my site. Now that I am authenticated and have a context to my SharePoint Online site, I'll run the cmdlet Get-SPOProvisioningTemplate, and give it the path to save the template file. You'll notice that I didn't need to explicitly specify the URL of the site to use for generating the template. By default, this is based on the context that was set from the URL I specified when I called Connect-SPOnline. But I could also specify a different URL using the -web switch. While this runs you'll see it begins by extracting the contents of the site, as well as its regional settings, security, fields, content types, list instances, features, and more, all of which form the basis of what is contained within the template file. Now that that's finished, let's take a look at the contents of the template.xml file we just generated. As you can see, there is a lot in here, but let's focus on the specific customizations that are of interest to us. Here in the pnp:WebSettings element, we see references to the SiteLogo and AlernateCSS properties we set for the site back in module 2. Notice the use of the site token with curly braces in these values. When this template is applied to our destination site, the engine will automatically replace this token anywhere it appears within the template file with the correct site URL to reference the appropriate location in the destination site. There are several other of these tokens that you can use, including things like sitecollection, listid, listurl, and masterpagegallery. You'll see me use the site token again when I make some manual updates to this template by hand. Moving down in the pnp:ContentTypes section, we see our custom Weather content type with its associated fields based on the custom site columns we created earlier. Scrolling down to the pnp:Lists element, we see a pnp:ListInstance entry for our WeatherList, including the binding to our custom content type, whose ID you see here. Here also we see the view fields and list columns associated with that list, so we know that will all be created in the destination site when we apply the provisioning template there. Scrolling down further you'll see under the pnp:CustomActions element, we have one custom action, which is a script link action that corresponds to the JavaScript we embedded on every page using the core.embed JavaScript add-in from module 2. You'll see the script block attribute contains the actual JavaScript markup we embedded as a string value, so that customization will be applied to the destination site as well. Note that we will need to update this value by hand to reflect the URL of the destination site, which I will do shortly when I make a couple of other changes. If we were to apply this provisioning template in its present state, our site columns, content type, and list instance without any list items would be successfully created in the destination site, but our site logo, alternate CSS file, and some other images and JavaScript files that we need to have deployed to the Site Assets directory would not. If we are only concerned about deploying the site logo and alternate CSS files, we can add the switch -PersistBrandingFiles in our call to Get-SPOProvisioningTemplate. This will add entries to the provisioning template that tell the provisioning engine to remote provision the files necessary to apply these user interface customizations on the destination site. I'll go ahead and re-run the cmdlet to generate our provisioning template. This time I'll add the switch to persist the branding files to show you what I'm talking bout. You'll notice the process saved the new template file, as well as the site logo and alternate CSS files to my desktop here. If we look at the template now, we'll see that a new pnp:Files element has been added, and includes items for our site logo and custom CSS file. Adding the -PersistBrandingFiles switch may be enough to meet your needs, if those are the only user interface customizations you need to apply. But keep in mind the site that we are using also requires a custom site background image, custom header, and web part JavaScript files, and all the forecast icon images used by our app script part in the Site Assets directory. Fortunately, although the built-in PnP PowerShell cmdlets may fail to capture a specific customization, keep in mind that you have complete control over this template file. Although you've seen me generate a template file using the built-in PowerShell cmdlets a couple times already, remember that you are free to create or update a provisioning template file by hand. You can also write code using the client object model to read, parse, and apply a provisioning template in a destination site. After reviewing this updated template, I need to make the following additions and updates to address our needs. One, I need to add list items to the list instance for WeatherList, reflecting the items currently in that list on the source site. Two, I need to add file entries for the other items from the source site's Site Assets directory that need to be provisioned to the destination site. And three, I need to update the site URL reference for the script link custom action. While I'm updating that I'm also going to modify the script link to include a reference to the forecast.js file that we used in our app script part from module 2. I'll make this change in conjunction with adding the forecast content div to the contents of the home.aspx page. These changes will allow us to encapsulate the forecast app script part without requiring any app parts or app script parts to be deployed to the destination site. I've gone ahead and made these updates. Notice how I have updated the ListInstance element for WeatherList to now include a collection of pnp:DataRow items, each containing pnp:DataValue items that map to the values stored in each field of the item. When the template is applied to the destination site, the provisioning engine will now add these items to the new list instance for me, automatically. If we look at the custom action element, I'll scroll over to show you how I have cleaned up this markup, as well as added the site token in curly braces, so that the correct site URL is referenced when loading the JavaScript files. I have also added a reference to the forecast.js file, so that script will now be included as part of the custom action as well. Scrolling down further, you'll see I've added entries for the other images and JavaScript files that need to be provisioned to the Site Assets library. The provisioning engine looks in the same directory on the file system as the template file, when provisioning these assets, so I've gone ahead and saved copies of all these files to my desktop, alongside my template file. The site logo and alternate CSS files referenced at the top of this list were already saved here by virtue of the -PersistBrandingFiles switch I used when running the Get-SPOProvisioningTemplate cmdlet. Finally you'll see that I have updated the contents of the WikiField field in the home.aspx page to include markup for a div called forecastContent, which the forecast.js file looks for when setting the markup for the app script part. With all of these updates made, I'm ready to apply this provisioning template to a brand new site. I created a new site collection under the path sites/test, which is what you see here. I'll go ahead and launch the SharePoint Online Management Shell again, and this time when I call Connect-SPOnline, I'll give the URL to this new site. After signing in, I'll run the cmdlet Apply-SPOProvisioningTemplate, and provide the path to my updated template XML file. As this runs, the provisioning engine is iterating through the template file to remote provision all the files and customizations we have specified. With that done, I'll go ahead and refresh the page. You will see all of our customizations have successfully been transferred to this site, including all of our user interface and declarative customizations. Best of all, we can reuse this provisioning template to apply these customizations to any number of destination sites, without any additional effort on our part, all with no additional add-ins to configure or install. And again, all of these customizations have been applied without requiring any dependencies on deployed solutions or the legacy Feature framework.

Review

In this module, you have seen the value of remote provisioning declarative artifacts such as site columns, content types, and list instances. For declarative customizations, you should favor techniques involving remote provisioning using the client object model to avoid creating permanent dependencies on specific features installed via the legacy Feature framework that may not be installed in a particular environment. This approach will make it easier for you to apply incremental updates as changes are needed down the road. We also took a look at the PnP Provisioning engine and how it can be used to capture and apply customizations made to a site using a provisioning template. We used PowerShell to create a provisioning template file based on our SharePoint Online site, made some additional updates by hand, and then applied it to a new site. We have now successfully migrated all user interface and declarative customizations from our on-premises farm to SharePoint Online, while at the same time eliminating the need for numerous farm and sandboxed solutions. In module 4, you will see how we can complete our transition to the cloud by replacing our on-premises timer job customization using Add-in model techniques.

Moving Timer Job Customizations to the Add-in Model

Introduction

Hi, this is Danny Jessee, and I would like to welcome you to module 4 of this course, Moving Timer Job Customizations to the Add-in Model. In this module, we will learn how back-end SharePoint processes currently implemented as timer jobs can be implemented using Add-in model techniques. Time jobs are configured to execute on a schedule, and automate back-end processes that operate on data stored in SharePoint sites. They are useful for performing a wide variety of tasks ranging from basic site governance to more resource-intensive operations that are best suited to being run outside of peak farm usage hours. Perhaps not surprisingly, the code you write to replace a custom timer job with what is known as a remote timer job, will require the use of the SharePoint client object model, and will be packaged in an application that runs on a server external to SharePoint, because this application will need to run in an automated manner on a schedule, without user intervention. We will also discuss how to properly and securely handle authenticating your remote time job. We will introduce the Office Developer PnP timer job framework, which like other PnP solutions we have seen in this course, provides a supported, repeatable approach to developing remote timer jobs. We will then look at options for deploying our remote timer jobs by leveraging an on-premises server with the Windows Task Scheduler, as well as a pure cloud-based solution that utilizes Azure WebJobs. In either case, our overarching objective is to develop an executable application that talks to SharePoint using the client object model and is deployed to a platform external to SharePoint that allows this application to be run in an automated manner on a defined schedule.

Externalizing Timer Job Logic

SharePoint timer jobs are deployed as farm solutions with web application-scoped features using the legacy Feature framework, running full trust code that inherits from the SPJobDefinition class. Unlike other full trust code we have seen in this course that runs in the same IIS worker process at SharePoint, timer jobs are executed in the SharePoint timer service process, owstimer.exe, and can be configured to run on servers in the farm other than front-end servers. However, because this custom code is still running in a server process somewhere within the SharePoint farm, the same risk exists of a poorly-coded timer job, potentially destabilizing the farm. Likewise, because timer jobs must be deployed as farm solutions, they cannot be deployed to SharePoint Online. While moving to the cloud removes numerous administrative burdens and helps to alleviate many others, plenty of organizations still require the functionality provided by timer jobs to address their business needs. To constitute a viable alternative to legacy timer jobs, the remote timer jobs we develop must be able to do all the same things that legacy timer jobs do, including having the ability to run autonomously on a schedule. Custom timer jobs in SharePoint on-premises are configured to run on a defined schedule, and SharePoint is responsible for handling the scheduling, as well as the execution of each timer job instance. With remote timer jobs, we will be responsible for this. In this module, we will look at two different approaches to automate the execution of our remote timer jobs on a schedule, one that leverages the built-in Windows Task Scheduler, and one that harnesses the power of the cloud using Azure WebJobs. Legacy timer jobs run in the context of the SharePoint farm account, so by definition they are able to access any resources they need. This makes them ideal candidates for administrative tasks that will always require the greatest level of permissions to run. Any remote timer jobs we develop must be explicitly granted permission to access and manipulate whatever SharePoint data is necessary for the timer job to function properly, or it must run in the context of a user account that has the necessary permissions. We'll see some different approaches for handling authentication and remote timer jobs later in this module. Legacy timer jobs can be scoped to one or more web applications, and often process data by iterating through a series of site collections and sub sites. Our remote timer jobs must also be capable of running against multiple sites when they execute. For historical information and debugging purposes, remote timer jobs must be able to log information about the actions they take and any errors they encounter, and store this log data in a location that is readily accessible to administrators. When we talk about externalizing timer job logic to use Add-in model techniques, we are of course referring to moving custom server-side code out of SharePoint. Based on everything you have learned so far, it should come as no surprise that in order to do this, you will need to leverage the SharePoint client object model in any remote timer job solution you develop. We've already seen several occasions throughout this course where custom server-side code could be rewritten to use the client object model instead. Through the use of the client object model's ClientContext object, it is possible to obtain a reference to any SharePoint site along with its lists, libraries, list items, and sub sites. As you have also seen in this course, we can compile our code that uses the .NET managed client object model into an executable Windows console application that runs from any location external to the SharePoint server. So we know how we will write the code to externalize and package our timer job logic, but how do we handle storing the credentials for the privileged account our remote timer job will need to execute as? In the examples we have seen so far, account credentials were stored in plain text in the source code. This is certainly not an advisable approach. In other instances we were able to prompt the user to enter credentials interactively, but of course this is not an option since timer jobs must run without any user interaction. We will examine some strategies to handle timer job credentials and authentication, more appropriately in production environments.

Using the Windows Task Scheduler

In this demo, we will create a simple console application to automate our on-premises timer job process, using the .NET managed client object model. We will then configure it to run as a scheduled task using the Windows Task Scheduler of an on-premises web server. In module 1, we introduced a timer job customization here in Central Administration of our on-premises farm. This timer job, called \_Update City Forecasts Timer Job, is responsible for creating new items in our WeatherList on a daily schedule. The meat of our on-premises timer job logic lies here in the Execute method of our class that inherits from the SPJobDefinition class defined in the server object model. As you can see, the job begins by getting a reference to the parent web application, as timer jobs are deployed as web application-scoped features. From there, it gets a reference to the RootWeb of the first site collection in the web application to access the WeatherList, where we then add items to that list in a loop. For demonstration purposes in this course, we're just generating some random temperature and condition values, but you can imagine a scenario where a custom timer job might connect to an external system to update data that is of value to your organization. Keep in mind we didn't need to do anything special to ensure this timer job had access to the list, or permission to create new items in it. Since legacy timer jobs run in the context of the farm account, it will automatically have permission to access any resources it needs. Unfortunately that will not be the case when we use Add-in model techniques, so we will need to keep that in mind when we develop our remote timer job replacement. Remember our objective is to develop a console application that uses the SharePoint client object model to replace the server-side code you see here. Running this console.exe program will then be doing the same thing as when the Execute method runs in our on-premises timer job. I've created a new console application and added a reference to the PnP Core component as you have seen me do before. In the Main method I'll go ahead and add the client object model code to replicate the functionality of our on-premises timer job. Notice that we are specifying our SiteUrl, as well as the UserName and Password of an account with privileges to access the site and update the WeatherList, using AppSettings values in our console application's AppConfig file. For Office 365, the credentials you specify here could be for an administrator account or a dedicated service account you create expressly for the purpose of running remote timer jobs. Keep in mind, however, that in addition to this information being stored in plain text in the configuration file, you must always ensure the user specified here always possesses the necessary permission to access whatever data is required by the remote timer job. Later in this module I'll show you an approach that alleviates this problem by allowing us to grant permissions directly to our application, without requiring a username and password that is tied to a user account. As you can see, our client object model code deletes the existing items in the list, then adds new ones. We'll be able to verify this worked by refreshing the page that contains our forecast web part, and seeing the values change along with the last updated date. I'll go ahead and run the application, then refresh the page, and you will see that the values were indeed updated. Now that we know our remote timer job logic is good to go, we need to go about automating the execution of this application. Fortunately we can take advantage of the Task Scheduler that is built in to Windows to do this. If I go to Start, then Administrative Tools, then Task Scheduler, the Task Scheduler here on my local server will launch. I'll select Task Scheduler Library, then right-click and select Create Basic Task. We can give our task a name and press Next. Now we can set the schedule for our task, which will be Daily, and then we can set a time for our task to run and how frequently the tasks should recur. To match what we had in Central Administration for our on-premises timer job, I'll have it run at midnight and recur every 1 day. Next we tell the Task Scheduler that we want to start a program every time the task runs. From here we can point it to our console application executable that we just built. On the Summary screen, we can verify all the options we just set, and then click Finish. Now our remote timer job will run automatically every day at midnight, much like our on-premises timer jobs that can be run on demand. I can also right-click this task and select Run, and the task will run immediately. We can see the history of the job by selecting the task and navigating to the History tab, and if I refresh the page here, you'll see the value is updated again from running the task just now. We have successfully replaced our on-premises timer job customization with the remote timer job that uses the client object model to update SharePoint Online without any farm solutions or server-side code. However, the approach you've just seen is not without its flaws, and it may not be the best approach for your environment. We had to store the credentials for a privileged user account in the application's configuration file. We don't have a clean way to scale this up to run against multiple different sites. There's no easy way for us to see any logging or debugging output that our application may generate, and we're still dependent on some on-premises infrastructure to have a server where we can deploy our remote timer job and run it via the Windows Task Scheduler. Later in this module, we will see a different approach that allows you to remove these barriers, and also deploy your remote timer jobs to the cloud, removing your on-premises infrastructure burden entirely.

PnP Timer Job Framework

In the previous demo, you saw an effective, albeit rudimentary way to externalize the logic in a timer job into a console application using the client object model, then use the Windows Task Scheduler to automate the scheduling and execution of that application. This is a perfectly valid approach and may be suitable enough to meet your needs. However, if your organization needs to migrate a large number of complex custom timer jobs, you are probably hoping for a more robust framework that will help you migrate these customizations to the Add-in model. This is where the Office Developer PnP timer job framework comes into play. Much like your custom on-premises timer jobs are developed by inheriting from the SPJobDefinition class of the server object model, your remote timer jobs will inherit from the TimerJob class defined in the PnP Core component. Implementing a remote timer job using this class is simple. Start by defining a class that inherits from this abstract base class with a function that handles the predefined event, TimerJobRun. Instantiate your remote TimerJob class from the main function in a console application, set up the authentication for the job, define the list of sites where the job needs to be executed, then call the Run function. Let's discuss each of these steps in a little more detail. Here is how your remote timer job class will look when using the PnP timer job framework. You'll include a using statement to reference OfficeDevPnP.Core.Framework.TimerJobs, then you will define your class to inherit from the abstract base class TimerJob, which is defined in the PnP Core component. In your class's constructor, set up a delegate to handle the TimerJobRun event. Notice one of the arguments here is of type TimerJobRunEventArgs. This object contains a property called WebClientContext, which is a client context object that is already initialized to the site where your timer job will be running. Of course this object has a web property, which corresponds to the clientcontext.web object we use to get a reference to the list our timer job needs to update. Inside this delegate is where you will implement the logic for your remote timer job, instead of directly within the program's Main method, as you saw me do in the previous demo. When using the PnP timer job framework, we use the Main method of our program to instantiate our remote timer job class, set up its authentication, and define the list of sites where the job needs to run before raising the TimerJobRun event, which executes the job. Notice in this example we are calling the method UseOffice365Authentication, and providing a username and password. Behind the scenes this approach is fundamentally the same as the one we used in the previous demo, and still requires the use of a privileged user account whose credentials must be stored somewhere. To avoid this, we will create what is known as an app principal with client ID and secret values, which allows us to assign permissions directly to our application rather than a specific user account, then we can replace the call to UseOffice365Authentication with UseAppOnlyAuthentication instead. When setting up a remote timer job, one of the most important decisions you will need to make is how it will be authenticated. You must choose between authenticating with or without user credentials, that is to say between using an app principal or a user principal as the identity for any requests made to SharePoint via the client object model. Whereas a user principal needs to be explicitly granted permissions to each site where your remote timer job may need to run, an app principal can be granted tenant-scoped permissions, which is most closely analogous to how legacy timer jobs run in the context of the SharePoint farm account. Unlike the call to UseOffice365Authentication or UseNetworkCredentialsAuthentication for on-premises scenarios, both of which take a username and password as parameters, the call to UseAppOnlyAuthentication takes client ID and secret values associated with an app principal as parameters instead. We can create and authorize an app principal by visiting a couple of pages in SharePoint, AppRegNew.aspx and AppInv.aspx. AppRegNew.aspx allows you to register a new app principal by either specifying or generating new client ID and secret values, which are the same values you will then use in your call to use app-only authentication. After registering your app principal, AppInv.aspx allows you to specify and grant the permissions your app principal will need. In the next demo, I'll show you how we can generate client ID and secret values for a new app principal that our remote timer job application will use and grant it the permissions it needs. After authenticating our remote timer job, the next thing we need to do is specify a list of one or more sites where the job needs to run. The PnP timer job framework provides us with methods to accomplish this in a couple of different ways. The first method, AddSite, is the one you will most likely use. In your application's Main method, you can make as many calls to AddSite as you need to specify each of the sites your remote timer job needs to process. You can also call ClearAddedSites to remove all sites that may have been previously added. Within your remote timer job class, you also have the option of overriding a method called UpdateAddedSites. This method takes a parameter that contains a list of strings corresponding to all of the sites that were passed in via calls to AddSite at the time the TimerJobRun event was raised. Within this method, you are free to add, update or delete entries from this list of strings as you see fit, giving you the final say when it comes to the sites your remote timer job runs against. The PnP timer job framework allows us to use a single wildcard asterisk at the end of any URL strings we specify. This enables us to specify entire collections of sites that match a certain pattern, without requiring multiple calls to AddSite. This is useful in cases where the remote timer job needs to process all site collections beneath a particular managed path. One caveat if you choose to make use of this wildcard: If your remote timer job uses app-only authentication, you will also need to specify something called enumeration credentials, which correspond to a username and password associated with a privileged account. This is unfortunate, but necessary, because some of the APIs used by the wildcard matching algorithm require user credentials to be specified. To avoid the need to specify any credentials when using app-only authentication, do not use a wildcard in your site URL strings.

Using the PnP Timer Job Framework with Azure WebJobs

In this demo, we will use the PnP timer job framework to develop a more robust alternative to the remote timer job we created earlier in this module, without requiring any user credentials. We will then deploy and configure this remote timer job as an Azure WebJob, so that it too can run in the cloud without any on-premises infrastructure. In order to leverage app-only authentication, we must first visit the AppRegNew.aspx page I mentioned earlier. The full path to that page starts at the site level, then \_layouts/15/AppRegNew.aspx. This page allows me to register a new app principal, which can then be granted permissions to access resources in SharePoint. When we configure our remote timer job, we will tell it to authenticate using client ID and secret values, instead of a username and password. We can either specify these values ourselves, or generate new ones. I'll go ahead and click the Generate buttons to populate the client ID and secret values, then I'll enter a title for my app principal. We must also specify values for app domain and redirect URI. These values are not used when leveraging the client object model from an external application, so we can just enter localhost and <https://localhost> so that these fields aren't blank. Press Create, and then you will see a confirmation screen. Take note of these client ID and secret values, as we will need to update our remote timer job's App.config file to reference them. Before we update our code, let's go ahead and grant our app principal the permissions it will need. To do that, I'll visit AppInv.aspx. I can look up my app by entering its client ID value here and pressing Lookup. You'll see it populates with the info we submitted on AppRegNew.aspx. Now we need to specify the permission request XML, which defines the permissions our app principal will need to be granted. I'll show you an easy way to generate this XML, if you don't have it already, from within Visual Studio. In Visual Studio, you can either create a new SharePoint Add-in project, or open an existing one. Every SharePoint Add-in project contains an AppManifest.xml file. If we open that file here and navigate to the Permissions tab, you can specify the different permissions your app will request when it is installed. You saw me do this for the add-in containing the app part we developed in module 2. Here you see the option to allow the add-in to make app-only calls to SharePoint, which we will want to select if we plan to leverage app-only authentication. We will also select the Site Collection scope and request the FullControl permission. If I press F7, I'll see the actual XML for this file. We will want to capture the AppPermissionRequests element along with its children, then copy and paste it in the Permission Request XML box in AppInv.aspx. I'll go ahead and paste my permission request XML here, and then press Create. I am prompted to trust this app principal and grant it the permissions we just requested, which as you can see includes the ability to make app-only calls to SharePoint. I'll press Trust It. Now when our remote timer job app identifies itself to SharePoint, using these client ID and secret values, it will have all the permissions we just requested. Here we are back in Visual Studio. We are going to update the remote timer job application we created in the previous demo to leverage the PnP remote timer job framework. Since we already added a reference to the PnP Core component NuGet package, we don't need to worry about doing that. We do, however, need to make some updates to our App.config file. We can remove the entry for SiteUrl since we will now specify that by calling the AddSite method in our remote TimerJob class. We can also remove the entries for username and password, and replace them with entries for ClientId and ClientSecret, based on the values we just set using AppRegNew.aspx. I added a new class called ForecastTimerJob to my project, which inherits from the abstract base class timer job. Now I will set up the event handler for the TimerJobRun event, which Visual Studio will create for me here. Heading back to my Program.cs file, I can now strip all this logic from the Main method and move it to this event handler. Since the timer job framework is now responsible for managing the list of sites against which our timer job will run, as well as the authentication for our remote timer job, we can remove the code that reads the AppSettings values for our site URL and user credentials, as well as sets the credentials for our ClientContext object. Remember also that the TimerJobRunEventArgs parameter to this event handler contains its own ClientContext object that is already initialized to the site where our remote timer job is running, so we can now use that instead. I'll update the using statement here to leverage e.WebClientContext. Heading back to our Program.cs file, we can now update our Main method to instantiate our remote timer job class, read the client ID and secret values from App.config, and call UseAppOnlyAuthentication, add the site where our job needs to run, and then call the Run method. Before we deploy this remote timer job to Azure, let's quickly run it and confirm that everything works as expected. I'll refresh the page and we'll see that the values did indeed update. Now we can create our new Azure WebJob by right-clicking our project in Visual Studio and selecting Publish as Azure WebJob. We will tell the job to run on a schedule, recurring every 1 day at midnight. Now we must publish the assets for our application to Azure Web Apps. We'll select that as our published target. Of course in order to publish to Azure you must have an Azure account. For this demo, I'm signed into my personal Azure account where I've already created an Azure Web App for my remote timer job demo. You can create a new web app or select an existing one as your published target. I'll select this existing web app, press OK, and then press Publish. Visual Studio will now deploy the assets for our remote timer job to Azure. Once this is complete, we can verify everything was published successfully by logging into the Azure portal at portal.azure.com. If I select App Services, then my remote timer jobs app, I can go to WebJobs under Settings, select the job, and click Run. You'll notice the status value of the WebJob updates. I can refresh the status now and see that the job just completed, and if I go back to refresh our SharePoint Online page, we'll see these values updated again. Remember when we set up our console application to run as a Windows scheduled task in the previous demo, we had no way to view the output generated by the application when it ran. Here in Azure I can click on Logs, drill down to my remote timer job, and see the full console output generated by my application for any previous instance of the job. Here you see the Complete message I displayed via a console.write statement in my code. This is great for debugging, as well as capturing any errors that may arise when your WebJob is executed. The astute observers among you may have noticed that we see n/a displayed here under SCHEDULE. The schedule we defined for our WebJob when we deployed it from Visual Studio is actually configured within Azure Scheduler. If I bring up the Scheduler Job Collections and drill down, you'll see my remote timer job is indeed set up as a scheduler job that is configured to run every day. We have now successfully replicated the functionality of our on-premises timer job using the PnP timer job framework, without requiring any on-premises infrastructure. By using Azure WebJobs, we were able to deploy all the resources associated with our console application to the cloud, and by leveraging app-only authentication, we were able to authorize it to access data in SharePoint without the need for any user credentials. With this complete, we have now migrated the last bit of custom functionality from our on-premises environment to SharePoint Online using Add-in model techniques.

Review

In this module, you have seen how to externalize the logic previously implemented in custom SharePoint on-premises timer jobs, into what are known as remote timer jobs, using the .NET managed client object model. Remote timer jobs are implemented as standalone executable applications that are configured to run on a schedule in an automated manner. We introduced the PnP timer job framework, which like other PnP solutions we have seen, provides a standard, repeatable approach to implementing remote timer jobs using Add-in model techniques. You have seen how remote timer jobs can be authenticated with the user principal, which allows you to leverage the credentials of a privileged user or service account, or an app principal, which allows you to assign permission directly to your application without the need for user credentials. You have also seen how we can automate the scheduling and execution of our remote timer jobs, using the on-premises Windows Task Scheduler or Azure WebJobs in the cloud. Recall the original problem scenario that we introduced in module 1. Your organization is leaving its on-premises SharePoint farm behind and is moving to the cloud. You had more questions than answers. Your on-premises farm was full of custom code, implemented using numerous farm and sandboxed solutions. User interface customizations were hardcoded in custom master pages. Custom site columns, content types, and list instances were created using declarative markup deployed via sandboxed solutions, and had created permanent dependencies on the legacy SharePoint Feature framework. You even had a custom timer job that you had to manage through Central Administration. How could you take inventory of all these customizations? Could this new Add-in model you had heard about be flexible and powerful enough to allow you to migrate all of these customizations to the cloud? Yes! In this course you have learned that the Add-in model is capable and powerful enough to solve the kinds of problems that you solved with server-side code in the past. You have seen how techniques from the Add-in model can allow you to apply user interface customizations with a lighter touch, avoiding the need to create and maintain custom master pages, while also making it easier for you to deploy updates down the road. You haves seen how remote provisioning assets such as JavaScript and CSS files, as well as customizations such as site columns, content types, and list instances can help you remove dependencies on the legacy Feature framework, while also giving you complete control over the install, upgrade, and uninstall of your custom capabilities. Finally you have seen that you aren't alone in this journey. The Office Developer Patterns and Practices initiative provides you with extensive samples, scenarios, solutions, and guidance to assist with your migration to Office 365. In this course we utilized several solution starters from the PnP Team to accomplish tasks like deploying branding assets and embedding JavaScript on every page in a site. We also used the PnP Provisioning engine to capture all of the customizations made to a site in a reusable template file, then applied these customizations to a new site. We concluded by making use of the PnP timer job framework to migrate our legacy timer job to the Add-in model. We did it! We migrated all of our legacy on-premises SharePoint customizations and are ready to go in SharePoint Online. Thank you so much for watching this course. I hope you found it valuable, and that it makes your job a little easier. Please feel free to ask questions or leave any feedback you may have on the discussion board for this course, or reach out to my on Twitter @dannyjessee. Thanks again, and have a great day.