



UX Components in SharePoint 2013 and SharePoint Online solution pack

Microsoft Corporation

October 2014

**Applies to:** SharePoint 2013 and SharePoint Online

**Summary:** This solution pack includes code and documents that demonstrate and describe techniques for creating custom UX components in provider-hosted apps for SharePoint 2013 and SharePoint Online.

©2014 Microsoft Corporation. All rights reserved.

This document is provided "as-is." Information and views expressed in this document, including URL and other Internet website references, may change without notice. You bear the risk of using it.

Some examples are for illustration only and are fictitious. No real association is intended or inferred.

This document does not provide you with any legal rights to any intellectual property in any Microsoft product. You may copy and use this document for your internal, reference purposes. You may modify this document for your internal, reference purposes.

# Contents

The UX Components in SharePoint 2013 and SharePoint Online solution pack includes one document, with the sections listed in Table 1.

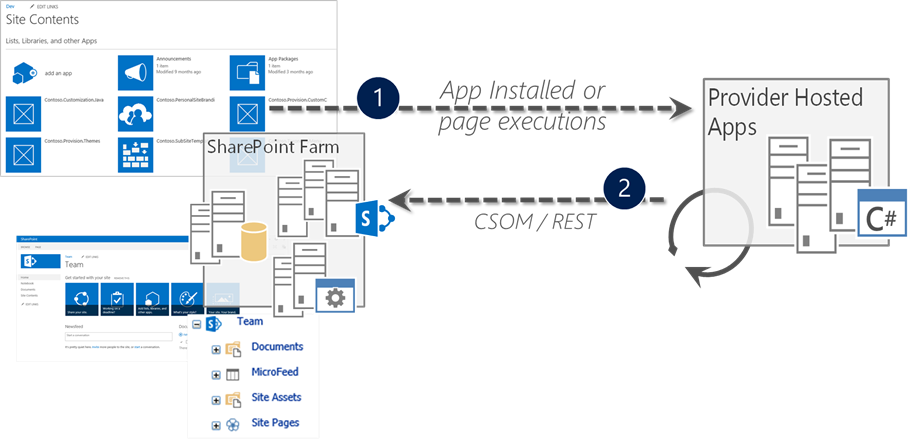
**Table 1. UX components in SharePoint 2013 and SharePoint Online solution pack modules**

|  |  |  |  |
| --- | --- | --- | --- |
| **Section** | **Name** | **Description** | |
| 2 | Controlling the UX with provider-hosted apps | Describes patterns for provider-hosted apps that create and modify SharePoint pages, layouts, and other SharePoint artifacts and that customize the user experience on host sites. | |
| 3 | Creating UX controls with provider-hosted apps | Demonstrates how to use JavaScript and the client object model (CSOM) to create UX controls for display in provider-hosted apps that interact with host sites. The examples include taxonomy picker, taxonomy menu, and people picker controls. | |
| 4 | Using OfficeWebWidgets | Demonstrates some ways to use the experimental [Office Web Widgets NuGet package](http://msdn.microsoft.com/en-us/library/office/dn636913(v=office.15).aspx) in provider-hosted apps. These include demonstrations of the people picker and list view widgets. | |
| 5 | Better performance through caching | Describes how to use HTML 5 and HTTP cookies to cache data and reduce the number of calls to SharePoint services. | |
|  | | |  |

# Controlling the UX with provider-hosted apps

The SharePoint 2013 app model provides a number of options for controlling and customizing the user experience on host sites by using the client object model (CSOM) running in a provider-hosted app. Figure 1 shows the two entry points for adding and modifying the SharePoint user experience through the app for SharePoint development model. You can add SharePoint components (lists, wiki pages, etc.) by handling the app installed event, and/or you can add and modify SharePoint components by using the CSOM.

**Figure 1. Deploying SharePoint components with the app model**



This section describes and links to samples that use both entry points and demonstrate best practices for addressing the following six scenarios for creating and customizing UX components remotely by using provider-hosted apps:

* Page manipulation (adding and modifying a wiki page)
* Showing apps and data in modal dialogs
* Personalized UI elements
* Client-side rendering (deploying JSLink files that customize the rendering of fields in SharePoint lists)
* Web part and app part manipulation (remotely provision and run an app script part in a provider-hosted app)
* Data aggregation and caching (using HTML 5 local storage and HTTP cookies to reduce the number of service calls to SharePoint)

## [Page manipulation](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.ModifyPages)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to add and modify content in wiki pages from a provider-hosted app. | This sample provides a convenient way to automate customizations on the host web, for example, whenever an app is installed or when a site is provisioned.  **Typical scenarios**:   * Create new pages * Apply page layouts and branding * Manipulate page elements (web parts, app parts, etc.) | This sample app uses the client object model (CSOM) to modify HTML and add web parts on pages in the host web. |

**Related samples**:

* [Provisioning.Pages](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Provisioning.Pages) (page manipulation)
* [Branding.ApplyBranding](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.ApplyBranding) (branding pages)

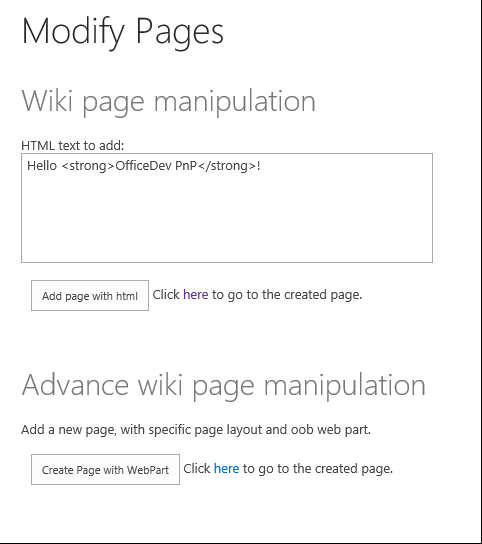
**Alternative approaches**:

* [Branding.CustomCSS](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.CustomCSS) (CSS injection to the host web)
* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)

This sample covers two broad scenarios: creating a wiki page and creating and then modifying the layout of a wiki page. This sample uses the default site pages library and existing “out-of-the-box” layouts, but it could be adapted to use a custom wiki page library and custom layouts.

The app’s UI presents you with two buttons that create both wiki pages, along with two links for viewing the wiki pages once you’ve created them (Figure 2).

**Figure 2. Page manipulation sample’s launch page**



The sample code for the first scenario checks to see if you’ve already created the wiki page, and if not, it adds the file to the site pages library and returns its URL.

var newpage = pageLibrary.RootFolder.Files.AddTemplateFile(newWikiPageUrl, TemplateFileType.WikiPage);

ctx.Load(newpage);

ctx.ExecuteQuery();

wikiPageUrl = String.Format("sitepages/{0}", wikiPageName);

In both scenarios, the sample adds the HTML entered via the text box on the launch page by using the AddHtmlToWikiPage method in a helper class called LabHelper. This method simply inserts the HTML from the form inside the WikiField field of the wiki page.

public void AddHtmlToWikiPage(ClientContext ctx, Web web, string folder, string html, string page)

{

Microsoft.SharePoint.Client.Folder pagesLib = web.GetFolderByServerRelativeUrl(folder);

ctx.Load(pagesLib.Files);

ctx.ExecuteQuery();

Microsoft.SharePoint.Client.File wikiPage = null;

foreach (Microsoft.SharePoint.Client.File aspxFile in pagesLib.Files)

{

if (aspxFile.Name.Equals(page, StringComparison.InvariantCultureIgnoreCase))

{

wikiPage = aspxFile;

break;

}

}

if (wikiPage == null)

{

return;

}

ctx.Load(wikiPage);

ctx.Load(wikiPage.ListItemAllFields);

ctx.ExecuteQuery();

string wikiField = (string)wikiPage.ListItemAllFields["WikiField"];

Microsoft.SharePoint.Client.ListItem listItem = wikiPage.ListItemAllFields;

listItem["WikiField"] = html;

listItem.Update();

ctx.ExecuteQuery();

}

The sample code for the second scenario creates a new WebPartEntity, and then uses methods in a helper class to populate the web part with XML that displays a promoted links list (consisting of <http://www.bing.com> and the home page of the OfficeDev/PnP GitHub repository).

WebPartEntity wp2 = new WebPartEntity();

wp2.WebPartXml = new LabHelper().WpPromotedLinks(linksID, string.Format("{0}/Lists/{1}",

Request.QueryString["SPHostUrl"], "Links"),

string.Format("{0}/{1}", Request.QueryString["SPHostUrl"],

scenario2PageUrl), "$Resources:core,linksList");

wp2.WebPartIndex = 1;

wp2.WebPartTitle = "Links";

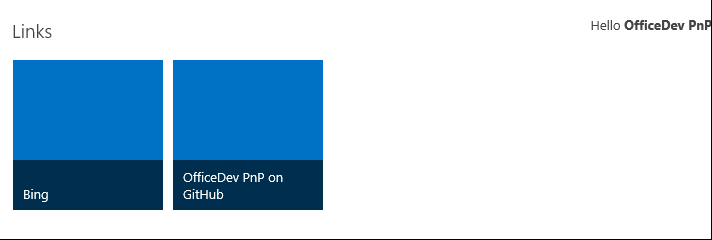
new LabHelper().AddWebPartToWikiPage(ctx, ctx.Web, "SitePages", wp2, scenario2Page, 2, 1, false);

new LabHelper().AddHtmlToWikiPage(ctx, ctx.Web, "SitePages", htmlEntry.Text, scenario2Page, 2, 2);

this.hplPage2.NavigateUrl = string.Format("{0}/{1}", Request.QueryString["SPHostUrl"], scenario2PageUrl);

The helper code displays the promoted links with a table inside an XsltListViewWebPart (Figure 3).

**Figure 3. Second wiki page with XsltListViewWeb part and promoted links table**



The LabHelper class’s WpPromotedLinks object contains XML that defines the appearance of the web part that will be injected into the wiki page.

The AddWebPartToWikiPage method then inserts the newly defined web part inside a new div tag on the wiki page.

XmlDocument xd = new XmlDocument();

xd.PreserveWhitespace = true;

xd.LoadXml(wikiField);

// Sometimes the wikifield content seems to be surrounded by an additional div?

XmlElement layoutsTable = xd.SelectSingleNode("div/div/table") as XmlElement;

if (layoutsTable == null)

{

layoutsTable = xd.SelectSingleNode("div/table") as XmlElement;

}

XmlElement layoutsZoneInner = layoutsTable.SelectSingleNode(string.Format("tbody/tr[{0}]/td[{1}]/div/div", row, col)) as XmlElement;

// - space element

XmlElement space = xd.CreateElement("p");

XmlText text = xd.CreateTextNode(" ");

space.AppendChild(text);

// - wpBoxDiv

XmlElement wpBoxDiv = xd.CreateElement("div");

layoutsZoneInner.AppendChild(wpBoxDiv);

if (addSpace)

{

layoutsZoneInner.AppendChild(space);

}

XmlAttribute attribute = xd.CreateAttribute("class");

wpBoxDiv.Attributes.Append(attribute);

attribute.Value = "ms-rtestate-read ms-rte-wpbox";

attribute = xd.CreateAttribute("contentEditable");

wpBoxDiv.Attributes.Append(attribute);

attribute.Value = "false";

// - div1

XmlElement div1 = xd.CreateElement("div");

wpBoxDiv.AppendChild(div1);

div1.IsEmpty = false;

attribute = xd.CreateAttribute("class");

div1.Attributes.Append(attribute);

attribute.Value = "ms-rtestate-read " + wpdNew.Id.ToString("D");

attribute = xd.CreateAttribute("id");

div1.Attributes.Append(attribute);

attribute.Value = "div\_" + wpdNew.Id.ToString("D");

// - div2

XmlElement div2 = xd.CreateElement("div");

wpBoxDiv.AppendChild(div2);

div2.IsEmpty = false;

attribute = xd.CreateAttribute("style");

div2.Attributes.Append(attribute);

attribute.Value = "display:none";

attribute = xd.CreateAttribute("id");

div2.Attributes.Append(attribute);

attribute.Value = "vid\_" + wpdNew.Id.ToString("D");

ListItem listItem = webPartPage.ListItemAllFields;

listItem["WikiField"] = xd.OuterXml;

listItem.Update();

ctx.ExecuteQuery();

## [Showing apps and data in modal dialogs](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.Dialog)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to display a provider-hosted app inside a modal dialog, inject links that launch the dialog, and use the JavaScript object model (JSOM) to get information from the host web and display it in a provider-hosted app. | This sample shows how to create custom actions, display provider-hosted apps inside dialog boxes that launch from links and/or custom actions, and use the cross-domain library to get content from the host web.  **Typical scenarios**:   * Display provider-hosted content in multiple display contexts. * Display host web data using JSOM. * Launch dialog boxes in multiple contexts. | This sample app uses the client object model (CSOM) to add custom actions and the JavaScript object model (JSOM), along with the [cross-domain library](http://msdn.microsoft.com/en-us/library/office/fp179927(v=office.15).aspx), to launch modal dialogs and display information from the host web. |

**Related samples**:

* [Core.JavaScriptCustomization](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.JavaScriptCustomization)

This sample demonstrates two methods for injecting modal dialog links that display a provider-hosted app page into a SharePoint host site. The app uses the client object model (CSOM) to create the custom action and JavaScript to launch and display information inside the dialog. Since some of this information comes from the host site, it also uses the JavaScript object model (JSOM) to retrieve information from the host site.

Also, since the app is running in a domain that is different from that of the SharePoint host site, it also uses the [SharePoint cross-domain library](http://msdn.microsoft.com/en-us/library/office/fp179927(v=office.15).aspx) to make the calls to the host site.

The launch page is also the page that displays in the dialog. In order to handle any differences in display that you might want to handle given the display context (dialog versus full-page), the app determines whether it’s being displayed in a dialog by using a query string parameter that is passed along with the links that launch the dialog boxes.

private string SetIsDlg(string isDlgValue)

{

var urlParams = HttpUtility.ParseQueryString(Request.QueryString.ToString());

urlParams.Set("IsDlg", isDlgValue);

return string.Format("{0}://{1}:{2}{3}?{4}", Request.Url.Scheme, Request.Url.Host, Request.Url.Port, Request.Url.AbsolutePath, urlParams.ToString());

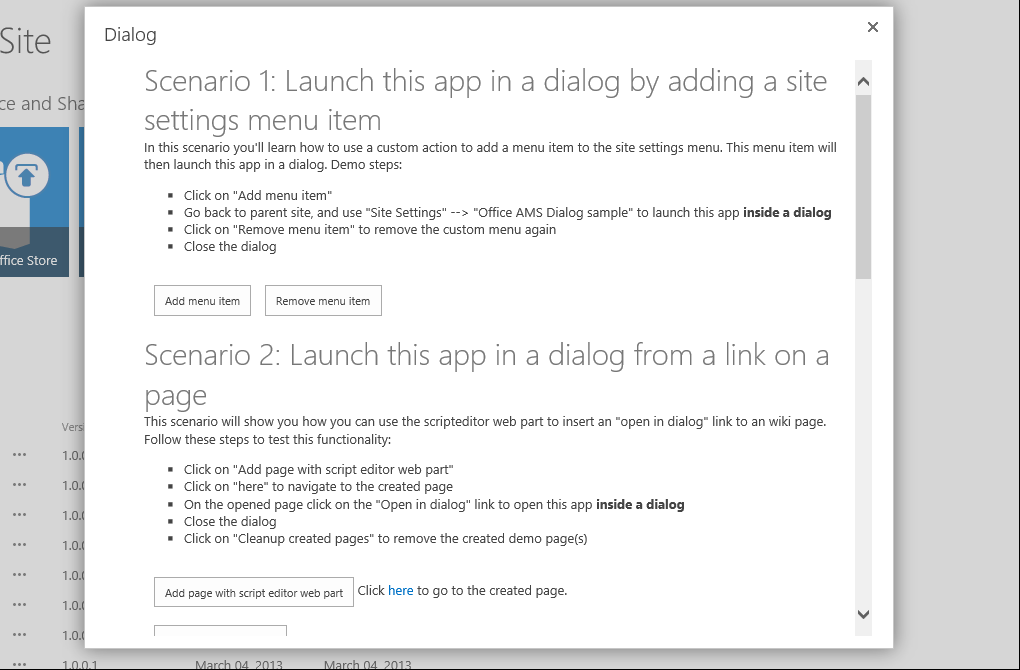
}

You could, for example, choose to display certain UI elements (buttons, etc.), or even different page layouts, depending on whether the content is being displayed in a dialog.

The launch page UI presents two options for creating links to the dialog, along with a list of all the lists on the host web. It also presents OK and Cancel buttons that you can use in the dialog box context to close the dialog and/or prompt additional actions in the app.

Figure 4 shows how this app appears inside a dialog box that has been launched from a custom action added to the Site Settings menu.

**Figure 4. Dialog sample app displayed inside a dialog box launched from the Site Settings menu**



When you choose the **Add menu item button**, the app creates a CustomActionEntity that contains JavaScript that launches the dialog box. It then uses the [OfficeDevPnP Core extension](https://github.com/OfficeDev/PnP/tree/dev/OfficeDevPnP.Core) method AddCustomAction to add the new custom action to the Site Settings menu.

StringBuilder modelDialogScript = new StringBuilder(10);

modelDialogScript.Append("javascript:var dlg=SP.UI.ModalDialog.showModalDialog({url: '");

modelDialogScript.Append(String.Format("{0}", SetIsDlg("1")));

modelDialogScript.Append("', dialogReturnValueCallback:function(res, val) {} });");

//Create a custom action

CustomActionEntity customAction = new CustomActionEntity()

{

Title = "Office AMS Dialog sample",

Description = "Shows how to launch an app inside a dialog",

Location = "Microsoft.SharePoint.StandardMenu",

Group = "SiteActions",

Sequence = 10000,

Url = modelDialogScript.ToString(),

};

//Add the custom action to the site

cc.Web.AddCustomAction(customAction);

Add AddCustomAction method adds the custom action to the UserCustomActions collection associated with the SharePoint site.

var newAction = existingActions.Add();

newAction.Description = customAction.Description;

newAction.Location = customAction.Location;

if (customAction.Location == JavaScriptExtensions.SCRIPT\_LOCATION)

{

newAction.ScriptBlock = customAction.ScriptBlock;

}

else

{

newAction.Sequence = customAction.Sequence;

newAction.Url = customAction.Url;

newAction.Group = customAction.Group;

newAction.Title = customAction.Title;

newAction.ImageUrl = customAction.ImageUrl;

if (customAction.Rights != null)

{

newAction.Rights = customAction.Rights;

}

}

newAction.Update();

web.Context.Load(web, w => w.UserCustomActions);

web.Context.ExecuteQuery();

When you choose the **Add page with script editor web part** button, the app uses the [OfficeDevPnP core](https://github.com/OfficeDev/PnP/tree/dev/OfficeDevPnP.Core) methods AddWikiPage and AddWebPartToWikiPage methods to create a wiki page inside the site pages library and add a configured script editor web part to the page.

string scenario1Page = String.Format("scenario1-{0}.aspx", DateTime.Now.Ticks);

string scenario1PageUrl = cc.Web.AddWikiPage("Site Pages", scenario1Page);

cc.Web.AddLayoutToWikiPage("SitePages", WikiPageLayout.OneColumn, scenario1Page);

WebPartEntity scriptEditorWp = new WebPartEntity();

scriptEditorWp.WebPartXml = ScriptEditorWebPart();

scriptEditorWp.WebPartIndex = 1;

scriptEditorWp.WebPartTitle = "Script editor test";

cc.Web.AddWebPartToWikiPage("SitePages", scriptEditorWp, scenario1Page, 1, 1, false);

The AddWikiPage method loads the site pages library, and if the wiki page specified by the app doesn’t already exist there, it creates the page.

public static string AddWikiPage(this Web web, string wikiPageLibraryName, string wikiPageName)

{

string wikiPageUrl = "";

var pageLibrary = web.Lists.GetByTitle(wikiPageLibraryName);

web.Context.Load(pageLibrary.RootFolder, f => f.ServerRelativeUrl);

web.Context.ExecuteQuery();

var pageLibraryUrl = pageLibrary.RootFolder.ServerRelativeUrl;

var newWikiPageUrl = pageLibraryUrl + "/" + wikiPageName;

var currentPageFile = web.GetFileByServerRelativeUrl(newWikiPageUrl);

web.Context.Load(currentPageFile, f => f.Exists);

web.Context.ExecuteQuery();

if (!currentPageFile.Exists)

{

var newpage = pageLibrary.RootFolder.Files.AddTemplateFile(newWikiPageUrl, TemplateFileType.WikiPage);

web.Context.Load(newpage);

web.Context.ExecuteQuery();

wikiPageUrl = UrlUtility.Combine("sitepages", wikiPageName);

}

return wikiPageUrl;

}

The AddWebPartToWikiPage method then inserts the newly defined web part inside a new div tag on the wiki page. This method is identical to the method of the same name used by the page manipulation sample in [section 2.1](#_Page_manipulation_(Core.ModifyPages).

The app retrieves the information that populates the list of SharePoint lists on the host web by using JSOM and the cross domain library.

function printAllListNamesFromHostWeb() {

var context;

var factory;

var appContextSite;

var collList;

context = new SP.ClientContext(appWebUrl);

factory = new SP.ProxyWebRequestExecutorFactory(appWebUrl);

context.set\_webRequestExecutorFactory(factory);

appContextSite = new SP.AppContextSite(context, spHostUrl);

this.web = appContextSite.get\_web();

collList = this.web.get\_lists();

context.load(collList);

context.executeQueryAsync(

Function.createDelegate(this, successHandler),

Function.createDelegate(this, errorHandler)

);

[How to: Access SharePoint 2013 data from apps using the cross-domain](http://msdn.microsoft.com/en-us/library/office/fp179927(v=office.15).aspx) library explains in detail how to use the cross-domain library to get information from the host web when your app is running JavaScript from a remote domain.

## [Personalized UI elements](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.UIElementPersonalization)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to use JavaScript injection, plus some values stored in user profiles and SharePoint lists to personalize user interface elements on the host web. It also uses HTML 5 local storage to minimize calls to the host site and therefore enhance performance. | This sample shows how to use a provider-hosted app to customize the user experience by looking up values stored in user profile information. It also shows how to use HTML 5 local storage to improve performance.  **Typical scenarios**:   * Display information targeted at specific users or types of users. * Minimize service calls to SharePoint. | This sample app deploys artifacts (images and a SharePoint list) to the host web, and then uses injected JavaScript to check HTML 5 local storage and user profile information. The JavaScript selects and displays one of the uploaded images on the home page of a SharePoint site. |

**Related samples**:

* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)
* [Performance.Caching](https://github.com/OfficeDev/PnP/tree/dev/Samples/Performance.Caching)

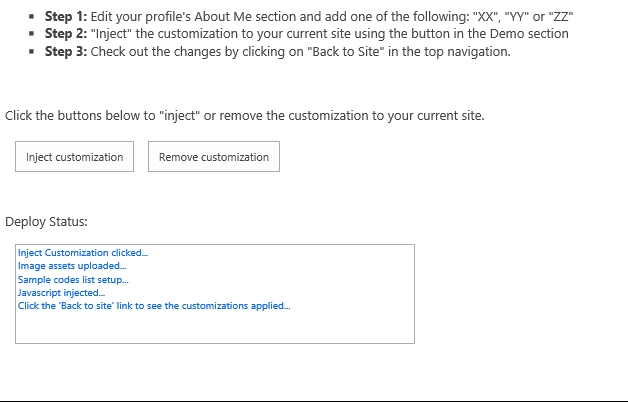
This sample displays one of three images on the home page of your SharePoint site, depending on user profile information. Figure 5 shows how your SharePoint site home page will look if you’ve placed the “YY” string in the **About Me** section of your user profile.

**Figure 5. Personalized UI element on SharePoint team site home page**



The launch page of this sample prompts you to add one of three strings – “XX”, “YY”, or “ZZ” -- to the **About Me** sections of your user profile page (Figure 6).

**Figure 6. Launch page instructions**



By this point the app has already deployed three images and a SharePoint list that contains titles and URLs for each image, along with an additional field that ties each image to one of the three strings. After you click on the **Inject customization** button, the app injects the personalize.js file into the user custom actions collection.

public void AddPersonalizeJsLink(ClientContext ctx, Web web)

{

string scenarioUrl = String.Format("{0}://{1}:{2}/Scripts", this.Request.Url.Scheme,

this.Request.Url.DnsSafeHost, this.Request.Url.Port);

string revision = Guid.NewGuid().ToString().Replace("-", "");

string personalizeJsLink = string.Format("{0}/{1}?rev={2}", scenarioUrl, "personalize.js", revision);

StringBuilder scripts = new StringBuilder(@"

var headID = document.getElementsByTagName('head')[0];

var");

scripts.AppendFormat(@"

newScript = document.createElement('script');

newScript.type = 'text/javascript';

newScript.src = '{0}';

headID.appendChild(newScript);", personalizeJsLink);

string scriptBlock = scripts.ToString();

var existingActions = web.UserCustomActions;

ctx.Load(existingActions);

ctx.ExecuteQuery();

var actions = existingActions.ToArray();

foreach (var action in actions)

{

if (action.Description == "personalize" &&

action.Location == "ScriptLink")

{

action.DeleteObject();

ctx.ExecuteQuery();

}

}

var newAction = existingActions.Add();

newAction.Description = "personalize";

newAction.Location = "ScriptLink";

newAction.ScriptBlock = scriptBlock;

newAction.Update();

ctx.Load(web, s => s.UserCustomActions);

ctx.ExecuteQuery();

}

Since SharePoint team sites by default use the [Minimal Download Strategy (MDS)](http://msdn.microsoft.com/en-us/library/office/dn456544(v=office.15).aspx), the code in the personalize.js file first attempts to register itself with MDS so that when you load the page that contains the JavaScript the MDS engine will launch the main function (RemoteManager\_Inject). If MDS has been disabled, this function is launched directly.

// Register script for MDS if possible

RegisterModuleInit("personalize.js", RemoteManager\_Inject); //MDS registration

RemoteManager\_Inject(); //non MDS run

if (typeof (Sys) != "undefined" && Boolean(Sys) && Boolean(Sys.Application)) {

Sys.Application.notifyScriptLoaded();h

}

if (typeof (NotifyScriptLoadedAndExecuteWaitingJobs) == "function") {

NotifyScriptLoadedAndExecuteWaitingJobs("scenario1.js");

}

The RemoteManager\_Inject function is the entry point for loading the other scripts that perform the customizations. When a given script depends on another script, be sure to load the dependent script after the one on which it depends. This sample loads the JQuery library before the personalizeIt function that uses JQuery.

function RemoteManager\_Inject() {

var jQuery = "https://ajax.aspnetcdn.com/ajax/jQuery/jquery-2.0.2.min.js";

// load jQuery

loadScript(jQuery, function () {

personalizeIt();

});

}

The personalizeIt() function checks HTML 5 local storage before looking up user profile information. If it goes to user profile information, it stores the information that it retrieves in HTML 5 local storage.

function personalizeIt() {

clientContext = SP.ClientContext.get\_current();

var fileref = document.createElement('script');

fileref.setAttribute("type", "text/javascript");

fileref.setAttribute("src", "/\_layouts/15/SP.UserProfiles.js");

document.getElementsByTagName("head")[0].appendChild(fileref);

SP.SOD.executeOrDelayUntilScriptLoaded(function () {

// Get localstorage values if they exist

buCode = localStorage.getItem("bucode");

buCodeTimeStamp = localStorage.getItem("buCodeTimeStamp");

// Check to see if the page already has injected personalized image

var pageTitle = $('#pageTitle')[0].innerHTML;

if (pageTitle.indexOf("img") > -1) {

personalized = true;

}

else {

personalized = false;

}

// If nothing in localstorage, get profile data, which will also populate localstorage

if (buCode == "" || buCode == null) {

getProfileData(clientContext);

personalized = false;

}

else {

// Check for expiration

if (isKeyExpired("buCodeTimeStamp")) {

getProfileData(clientContext);

if (buCode != "" || buCode != null) {

// Set timestamp for expiration

currentTime = Math.floor((new Date().getTime()) / 1000);

localStorage.setItem("buCodeTimeStamp", currentTime);

// Set personalized to false so that the code can check for a new image in case buCode was updated

personalized = false;

}

}

}

// Load image or make sure it is current based on value in AboutMe

if (!personalized) {

loadPersonalizedImage(buCode);

}

}, 'SP.UserProfiles.js');

}

The personalize.js file also contains code that local storage key expiration, since this isn’t built into HTML 5 local storage. This function checks to see whether the local storage key has expired. The HTML 5 local storage sample described in the [Performance caching section](#_Performance_caching) also uses this function.

// Check to see if the key has expired

function isKeyExpired(TimeStampKey) {

// Retrieve the example setting for expiration in seconds

var expiryStamp = localStorage.getItem(TimeStampKey);

if (expiryStamp != null && cacheTimeout != null) {

// Retrieve the timestamp and compare against specified cache timeout settings to see if it is expired

var currentTime = Math.floor((new Date().getTime()) / 1000);

if (currentTime - parseInt(expiryStamp) > parseInt(cacheTimeout)) {

return true; //Expired

}

else {

return false;

}

}

else {

//default

return true;

}

}

## [Client-side rendering](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.ClientSideRendering)

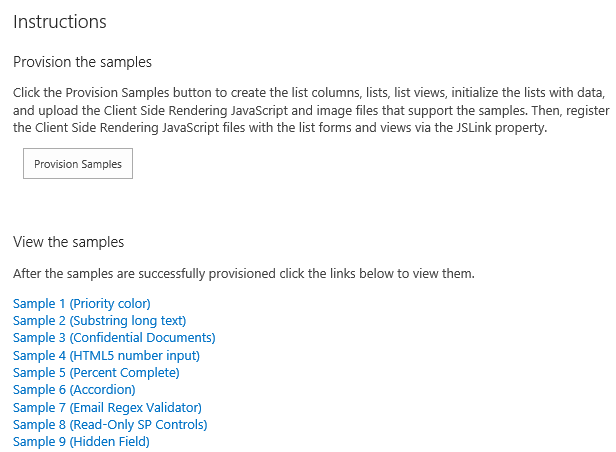
|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to use a provider-hosted app to remotely provision SharePoint artifacts and JSLink files that use client-side rendering to customize the look and behavior of SharePoint list fields. | The sample demonstrates how the remote provisioning pattern may be used to deploy client-side rendering components and associate them with views and forms in a SharePoint list. JSLink files and client-side rendering give you control over how controls on a SharePoint page (list views, list fields, add and edit forms, etc.) are rendered. Client-side rendering can reduce or eliminate the need for custom field types, since they make it possible to control list field appearance remotely. | This sample combines the JSLink samples from [Muawiyah Shannak's Client-side rendering (JSLink) code samples](http://code.msdn.microsoft.com/office/Client-side-rendering-JS-2ed3538a) into a single provider-hosted app for SharePoint that provisions the JSLink files. Client-side rendering enables you to use standard web technologies, such as HTML and JavaScript, to define the rendering logic of custom and predefined field types. |

**Alternative approaches**:

* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)

This sample app provides a convenient platform for demonstrating all of [Muawiyah Shannak's Client-side rendering (JSLink) code samples](http://code.msdn.microsoft.com/office/Client-side-rendering-JS-2ed3538a). The launch page prompts you to provision all of the samples (Figure 7).

**Figure 7. Launch page of client-side rendering sample**



When you click the **Provision Samples** button, the app deploys an image, all of the SharePoint lists, list views, list items, forms, and JavaScript files that are used in each sample. The app creates a folder named JSLink-Samples in the Style Library and then uploads the JavaScript files into that folder. The UploadFileToFolder method does the work of uploading and checking in each JavaScript file.

public static void UploadFileToFolder(Web web, string filePath, Folder folder)

{

using (FileStream fs = new FileStream(filePath, FileMode.Open))

{

FileCreationInformation flciNewFile = new FileCreationInformation();

flciNewFile.ContentStream = fs;

flciNewFile.Url = System.IO.Path.GetFileName(filePath);

flciNewFile.Overwrite = true;

Microsoft.SharePoint.Client.File uploadFile = folder.Files.Add(flciNewFile);

uploadFile.CheckIn("CSR sample js file", CheckinType.MajorCheckIn);

folder.Context.Load(uploadFile);

folder.Context.ExecuteQuery();

}

}

**Sample 1** shows how to apply formatting to a list column based on the field value. Priority field values of 1 (High), 2 (Normal), and 3 (Low) are displayed in red, orange, and yellow, respectively (Figure 8)

**Figure 8. Custom list field display**



.

The following JavaScript overrides the default field display creates a new display template for the Priority list field. The technique in the anonymous function that loads the contextual information about the field whose display you want to override is used in all of the samples.

(function () {

// Create object that has the context information about the field that we want to render differently

var priorityFiledContext = {};

priorityFiledContext.Templates = {};

priorityFiledContext.Templates.Fields = {

// Apply the new rendering for Priority field in List View

"Priority": { "View": priorityFiledTemplate }

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(priorityFiledContext);

})();

// This function provides the rendering logic for list view

function priorityFiledTemplate(ctx) {

var priority = ctx.CurrentItem[ctx.CurrentFieldSchema.Name];

// Return html element with appropriate color based on the Priority column’s value

switch (priority) {

case "(1) High":

return "<span style='color :#f00'>" + priority + "</span>";

break;

case "(2) Normal":

return "<span style='color :#ff6a00'>" + priority + "</span>";

break;

case "(3) Low":

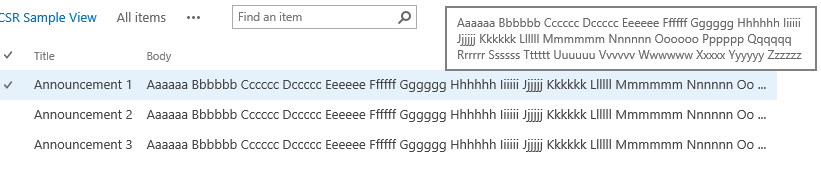
return "<span style='color :#cab023'>" + priority + "</span>";

}

}

**Sample 2** shows how to truncate long text stored in the Body field of an Announcements list. The complete text displays as a popup that appears whenever you hover over a list item (Figure 9).

**Figure 9. Truncated list field display with popup**



The following JavaScript truncates the Body field text and cause the full text to appear as a popup via the title attribute of the span tag.

function bodyFiledTemplate(ctx) {

var bodyValue = ctx.CurrentItem[ctx.CurrentFieldSchema.Name];

//This regex expression is used to delete html tags from the Body field

var regex = /(<([^>]+)>)/ig;

bodyValue = bodyValue.replace(regex, "");

var newBodyValue = bodyValue;

if (bodyValue && bodyValue.length >= 100)

{

newBodyValue = bodyValue.substring(0, 100) + " ...";

}

return "<span title='" + bodyValue + "'>" + newBodyValue + "</span>";

}

**Sample 3** shows how to display an image next to a document name inside a document library. A red badge appears whenever the Confidential field value is **Yes** (Figure 10).

**Figure 10. Image display next to document name**



The following JavaScript checks the Confidential field value and then customize the Name field display based on the value of another field. The sample uses the image that is uploaded when you click on the **Provision Samples** button.

function linkFilenameFiledTemplate(ctx) {

var confidential = ctx.CurrentItem["Confidential"];

var title = ctx.CurrentItem["FileLeafRef"];

// This Regex expression use to delete extension (.docx, .pdf ...) form the file name

title = title.replace(/\.[^/.]+$/, "")

// Check confidential field value

if (confidential && confidential.toLowerCase() == 'yes') {

// Render HTML that contains the file name and the confidential icon

return title + "&nbsp;<img src= '" + \_spPageContextInfo.siteServerRelativeUrl + "/Style%20Library/JSLink-Samples/imgs/Confidential.png' alt='Confidential Document' title='Confidential Document'/>";

}

else

{

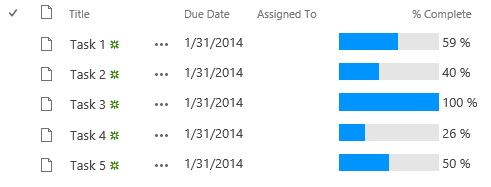
return title;

}

}

**Sample 4** shows how to display a “percent complete” bar chart in the % Complete field of a task list. The appearance of the bar chart depends on the value of the % Complete field (Figure 11).

**Figure 11. % Complete bar chart displays the value of the % Complete field**



A bar chart will also appear in the forms for creating and editing task list items.

The following JavaScript creates the bar chart display and associate it with the view and display forms (percentCompleteViewFiledTemplate) and then with the new and edit forms (percentCompleteEditFiledTemplate).

// This function provides the rendering logic for View and Display forms

function percentCompleteViewFiledTemplate(ctx) {

var percentComplete = ctx.CurrentItem[ctx.CurrentFieldSchema.Name];

return "<div style='background-color: #e5e5e5; width: 100px; display:inline-block;'> \

<div style='width: " + percentComplete.replace(/\s+/g, '') + "; background-color: #0094ff;'> \

&nbsp;</div></div>&nbsp;" + percentComplete;

}

// This function provides the rendering logic for New and Edit forms

function percentCompleteEditFiledTemplate(ctx) {

var formCtx = SPClientTemplates.Utility.GetFormContextForCurrentField(ctx);

// Register a callback just before submit.

formCtx.registerGetValueCallback(formCtx.fieldName, function () {

return document.getElementById('inpPercentComplete').value;

});

return "<input type='range' id='inpPercentComplete' name='inpPercentComplete' min='0' max='100' \

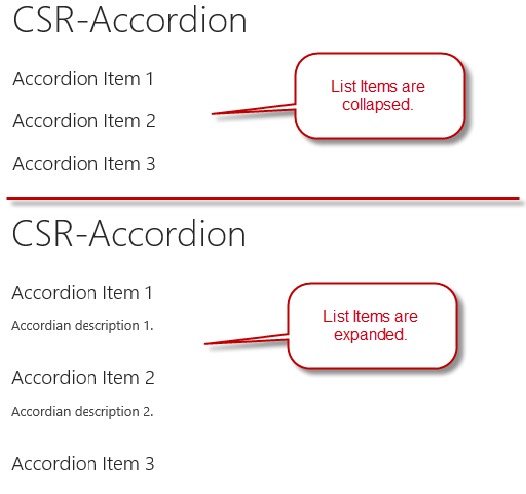
oninput='outPercentComplete.value=inpPercentComplete.value' value='" + formCtx.fieldValue + "' /> \

<output name='outPercentComplete' for='inpPercentComplete' >" + formCtx.fieldValue + "</output>%";

}

**Sample 5** shows how to change the rendering template for an entire list view. This view displays list item titles that expand like accordions when you click on them. The expanded view shows you additional list item fields (Figure 12).

**Figure 12. Collapsed and expanded list item views**



The following JavaScript sets up this template and registers it with the list template. It sets up the overall layout, and then uses the OnPostRender event handler to register the JavaScript function that executes when the list is rendered. This function associates the event with the CSS and click event handling that implements the accordion functionality.

(function () {

// jQuery library is required in this sample

// Fallback to loading jQuery from a CDN path if the local is unavailable

(window.jQuery || document.write('<script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-1.10.0.min.js"><\/script>'));

// Create object that has the context information about the field that we want to render differently

var accordionContext = {};

accordionContext.Templates = {};

// Be careful when adding the header for the template, because it will break the default list view render

accordionContext.Templates.Header = "<div class='accordion'>";

accordionContext.Templates.Footer = "</div>";

// Add OnPostRender event handler to add accordion click events and style

accordionContext.OnPostRender = accordionOnPostRender;

// This line of code tells the TemplateManager that we want to change all the HTML for item row rendering

accordionContext.Templates.Item = accordionTemplate;

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(accordionContext);

})();

// This function provides the rendering logic

function accordionTemplate(ctx) {

var title = ctx.CurrentItem["Title"];

var description = ctx.CurrentItem["Description"];

// Return whole item html

return "<h2>" + title + "</h2><p>" + description + "</p><br/>";

}

function accordionOnPostRender() {

// Register event to collapse and expand when clicking on accordion header

$('.accordion h2').click(function () {

$(this).next().slideToggle();

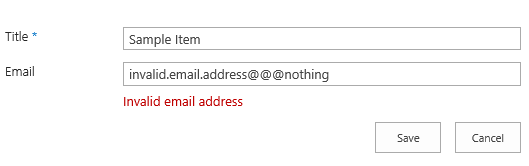
}).next().hide();

$('.accordion h2').css('cursor', 'pointer');

}

**Sample 6** shows how to use regular expressions to validate field values supplied by the user. A red error message appears when the user types an invalid email address into the Email field text box. This happens when the user either creates or edits a list item (Figure 13).

**Figure 13. Error message for invalid field text input**



The following JavaScript sets up the template with a placeholder for displaying the error message and register callback functions that fire whenever the user attempts to submit the new and edit forms. The first callback returns the value of the Email column, and the second callback uses regular expressions to validate that value.

function emailFiledTemplate(ctx) {

var formCtx = SPClientTemplates.Utility.GetFormContextForCurrentField(ctx);

// Register a callback just before submit.

formCtx.registerGetValueCallback(formCtx.fieldName, function () {

return document.getElementById('inpEmail').value;

});

//Create container for various validations

var validators = new SPClientForms.ClientValidation.ValidatorSet();

validators.RegisterValidator(new emailValidator());

// Validation failure handler.

formCtx.registerValidationErrorCallback(formCtx.fieldName, emailOnError);

formCtx.registerClientValidator(formCtx.fieldName, validators);

return "<span dir='none'><input type='text' value='" + formCtx.fieldValue + "' maxlength='255' id='inpEmail' class='ms-long'> \

<br><span id='spnError' class='ms-formvalidation ms-csrformvalidation'></span></span>";

}

// Custom validation object to validate email format

emailValidator = function () {

emailValidator.prototype.Validate = function (value) {

var isError = false;

var errorMessage = "";

//Email format Regex expression

var emailRejex = /\S+@\S+\.\S+/;

if (!emailRejex.test(value) && value.trim()) {

isError = true;

errorMessage = "Invalid email address";

}

//Send error message to error callback function (emailOnError)

return new SPClientForms.ClientValidation.ValidationResult(isError, errorMessage);

};

};

// Add error message to spnError element under the input field element

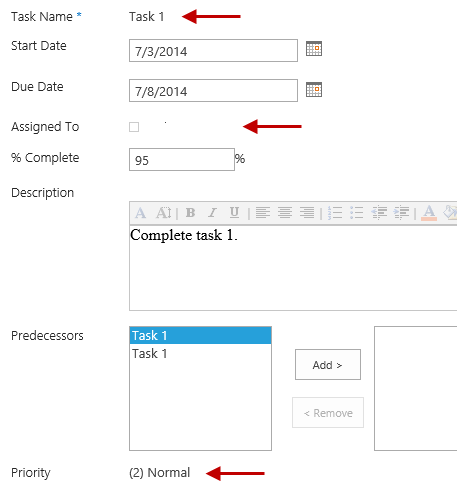
function emailOnError(error) {

document.getElementById("spnError").innerHTML = "<span role='alert'>" + error.errorMessage + "</span>";

}

**Sample 7** shows how to make fields read-only in list item edit forms. The read-only fields display with no editing controls (Figure 14).

**Figure 14. Read-only fields in a custom list edit form**



The following JavaScript modifies the display of the Title, AssignedTo, and Priority fields in the list item edit form so that they display the field values only with no editing controls. The code shows how to handle the parsing requirements for different field types.

function readonlyFieldTemplate(ctx) {

//Reuse SharePoint JavaScript libraries

switch (ctx.CurrentFieldSchema.FieldType) {

case "Text":

case "Number":

case "Integer":

case "Currency":

case "Choice":

case "Computed":

return SPField\_FormDisplay\_Default(ctx);

case "MultiChoice":

prepareMultiChoiceFieldValue(ctx);

return SPField\_FormDisplay\_Default(ctx);

case "Boolean":

return SPField\_FormDisplay\_DefaultNoEncode(ctx);

case "Note":

prepareNoteFieldValue(ctx);

return SPFieldNote\_Display(ctx);

case "File":

return SPFieldFile\_Display(ctx);

case "Lookup":

case "LookupMulti":

return SPFieldLookup\_Display(ctx);

case "URL":

return RenderFieldValueDefault(ctx);

case "User":

prepareUserFieldValue(ctx);

return SPFieldUser\_Display(ctx);

case "UserMulti":

prepareUserFieldValue(ctx);

return SPFieldUserMulti\_Display(ctx);

case "DateTime":

return SPFieldDateTime\_Display(ctx);

case "Attachments":

return SPFieldAttachments\_Default(ctx);

case "TaxonomyFieldType":

//Re-use JavaScript from the sp.ui.taxonomy.js SharePoint JavaScript library

return SP.UI.Taxonomy.TaxonomyFieldTemplate.renderDisplayControl(ctx);

}

}

//User control need specific formatted value to render content correctly

function prepareUserFieldValue(ctx) {

var item = ctx['CurrentItem'];

var userField = item[ctx.CurrentFieldSchema.Name];

var fieldValue = "";

for (var i = 0; i < userField.length; i++) {

fieldValue += userField[i].EntityData.SPUserID + SPClientTemplates.Utility.UserLookupDelimitString + userField[i].DisplayText;

if ((i + 1) != userField.length) {

fieldValue += SPClientTemplates.Utility.UserLookupDelimitString

}

}

ctx["CurrentFieldValue"] = fieldValue;

}

//Choice control need specific formatted value to render content correctly

function prepareMultiChoiceFieldValue(ctx) {

if (ctx["CurrentFieldValue"]) {

var fieldValue = ctx["CurrentFieldValue"];

var find = ';#';

var regExpObj = new RegExp(find, 'g');

fieldValue = fieldValue.replace(regExpObj, '; ');

fieldValue = fieldValue.replace(/^; /g, '');

fieldValue = fieldValue.replace(/; $/g, '');

ctx["CurrentFieldValue"] = fieldValue;

}

}

//Note control need specific formatted value to render content correctly

function prepareNoteFieldValue(ctx) {

if (ctx["CurrentFieldValue"]) {

var fieldValue = ctx["CurrentFieldValue"];

fieldValue = "<div>" + fieldValue.replace(/\n/g, '<br />'); + "</div>";

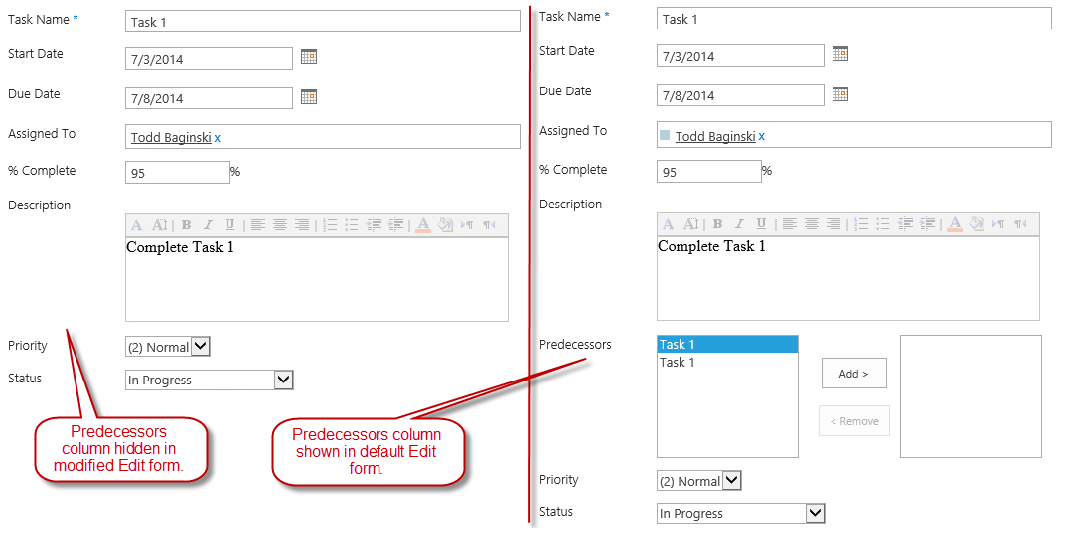
ctx["CurrentFieldValue"] = fieldValue;

}

}

**Sample 8** shows how to hide fields in list item new and edit forms. The sample hides the Predecessors field when a user creates or edits a task list item (Figure 15).

**Figure 15. Custom list item edit form hides the Predecessors field**



This sample deploys as the edit and new form for a list called **CSR-Hide-Controls list**. The [Readme file for this sample’s GitHub repository](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.ClientSideRendering) contains instructions for viewing this form after you’ve deployed the sample.

The following JavaScript finds the Predecessors field in the form’s HTML and hides it. The field remains present in the HTML, but the user can’t see it in the browser.

(function () {

// jQuery library is required in this sample

// Fallback to loading jQuery from a CDN path if the local is unavailable

(window.jQuery || document.write('<script src="//ajax.aspnetcdn.com/ajax/jquery/jquery-1.10.0.min.js"><\/script>'));

// Create object that has the context information about the field that we want to render differently

var hiddenFiledContext = {};

hiddenFiledContext.Templates = {};

hiddenFiledContext.Templates.OnPostRender = hiddenFiledOnPreRender;

hiddenFiledContext.Templates.Fields = {

// Apply the new rendering for Predecessors field in New and Edit forms

"Predecessors": {

"NewForm": hiddenFiledTemplate,

"EditForm": hiddenFiledTemplate

}

};

SPClientTemplates.TemplateManager.RegisterTemplateOverrides(hiddenFiledContext);

})();

// This function provides the rendering logic

function hiddenFiledTemplate() {

return "<span class='csrHiddenField'></span>";

}

// This function provides the rendering logic

function hiddenFiledOnPreRender(ctx) {

jQuery(".csrHiddenField").closest("tr").hide();

}

## [Web part and app part manipulation](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.AppScriptPart)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to use app script parts to embed scripts running in a provider-hosted app on a SharePoint page. How to make an app script part available from the web part gallery. | This sample shows how to modify the UI of a page on the host site by deploying an app script part and adding it to a SharePoint page from the web part gallery.  **Typical scenarios**:   * Display information from a user profile on a team site * Display remotely hosted information and functionality on a SharePoint site | The provider-hosted app runs JavaScript on a remotely hosted site. Users embed the app on a SharePoint page after finding it in the web part gallery. Since the app script part runs on a remote host, developers can update the look and feel of the app part remotely without having to any additional deployments or modifications to the SharePoint site. |

**Related samples**:

* [Branding.UIElementPersonalization](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.UIElementPersonalization)

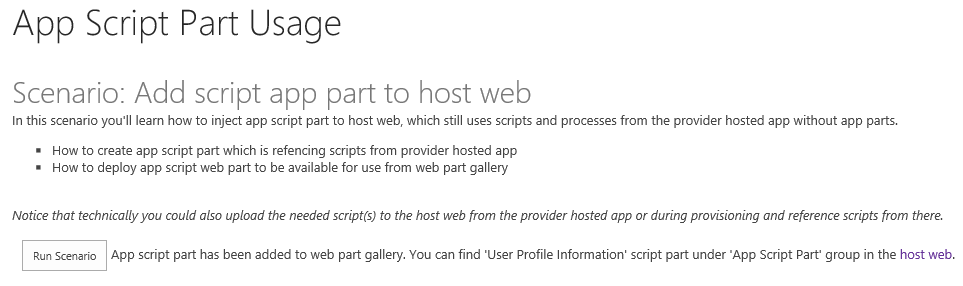
**Alternative approaches**:

* [Core.JavaScriptCustomization](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.JavaScriptCustomization)
* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)

This sample shows how to create and deploy an app script part. An app script part is like a web part in that you can add it to a SharePoint page from the web part gallery, but in this case the .webpart file simply embeds a JavaScript file that runs remotely in a provider-hosted app. The app script part runs inside a <div> tag on the SharePoint page and therefore provides a more responsive design and experience than you can get with app parts that run in IFrames.

The launch page presents you with a Run Scenario button that deploys the app script part to the web part gallery (Figure 16).

**Figure 16. The Run Scenario button deploys the app script part to the web part gallery**



The following code constructs a FileCreationInformationObject that contains the contents of the .webpart file and then uploads the new file into the web part gallery. You could also run this code automatically when the app part installs or as part of the site collection provisioning process. (See Module 5 of the Branding and Site Provisioning Solution Pack for guidance on automating steps as part of the provisioning process.)

var spContext = SharePointContextProvider.Current.GetSharePointContext(Context);

using (var clientContext = spContext.CreateUserClientContextForSPHost())

{

var folder = clientContext.Web.Lists.GetByTitle("Web Part Gallery").RootFolder;

clientContext.Load(folder);

clientContext.ExecuteQuery();

//upload the "OneDrive for Business Usage Guidelines.docx"

using (var stream = System.IO.File.OpenRead(Server.MapPath("~/userprofileinformation.webpart")))

{

FileCreationInformation fileInfo = new FileCreationInformation();

fileInfo.ContentStream = stream;

fileInfo.Overwrite = true;

fileInfo.Url = "userprofileinformation.webpart";

File file = folder.Files.Add(fileInfo);

clientContext.ExecuteQuery();

}

// Let's update the group for just uplaoded web part

var list = clientContext.Web.Lists.GetByTitle("Web Part Gallery");

CamlQuery camlQuery = CamlQuery.CreateAllItemsQuery(100);

Microsoft.SharePoint.Client.ListItemCollection items = list.GetItems(camlQuery);

clientContext.Load(items);

clientContext.ExecuteQuery();

foreach (var item in items)

{

// Just random group name to diffentiate it from the rest

if (item["FileLeafRef"].ToString().ToLowerInvariant() == "userprofileinformation.webpart")

{

item["Group"] = "App Script Part";

item.Update();

clientContext.ExecuteQuery();

}

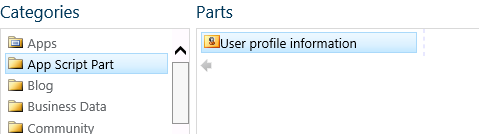
}

lblStatus.Text = string.Format("App script part has been added to web part gallery. You can find 'User Profile Information' script part under 'App Script Part' group in the <a href='{0}'>host web</a>.", spContext.SPHostUrl.ToString());

}

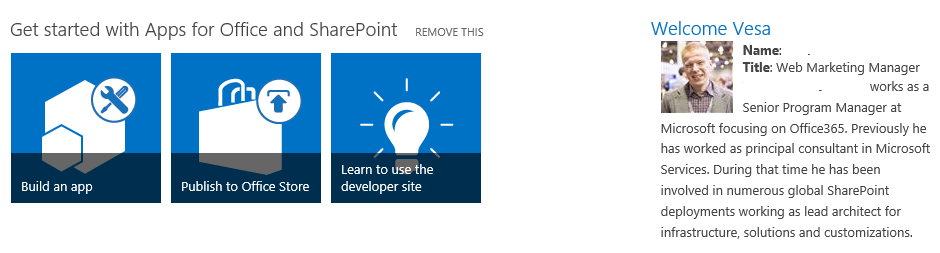
Once this step is complete, you’ll find the **User profile information** app script part inside a new **App Script Part** category in the web part gallery (Figure 17).

**Figure 17. The new App Script Part category in the web part gallery**



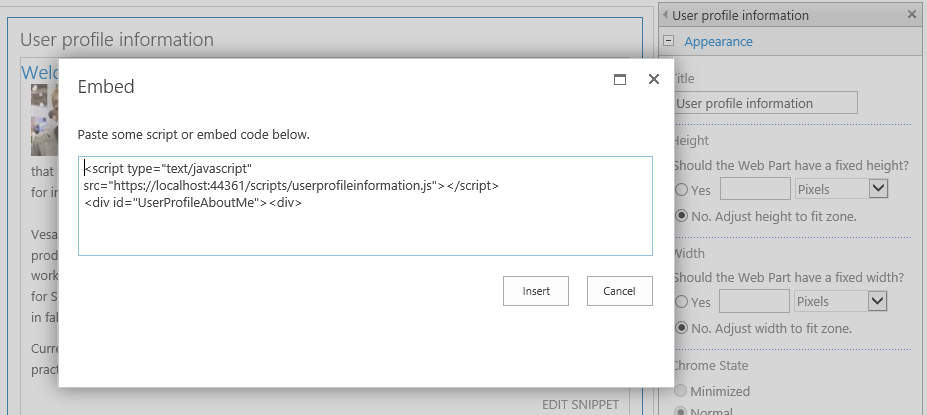
After you’ve added the app script part to the page, the remotely running JavaScript controls the display of the information on the page (Figure 18).

**Figure 18. The app script part runs remotely but controls its display on the SharePoint page**



When you view the app script part in edit mode, you’ll see that simply embeds the JavaScript file that is running remotely (Figure 19). In Figure 19, the script file runs on <https://localhost> because it is running in the F5 debugger in Visual Studio 2013.

**Figure 19. The app script part embeds a script that runs remotely**



The userprofileinformation.js script uses the JSOM to get user profile information from the host site. A more advanced implementation would enhance performance by using HTML 5 local storage, as in the sample described in [section 2.3](#_Personalized_UI_elements).

function sharePointReady() {

clientContext = SP.ClientContext.get\_current();

var fileref = document.createElement('script');

fileref.setAttribute("type", "text/javascript");

fileref.setAttribute("src", "/\_layouts/15/SP.UserProfiles.js");

document.getElementsByTagName("head")[0].appendChild(fileref);

SP.SOD.executeOrDelayUntilScriptLoaded(function () {

//Get Instance of People Manager Class

var peopleManager = new SP.UserProfiles.PeopleManager(clientContext);

//Get properties of the current user

userProfileProperties = peopleManager.getMyProperties();

clientContext.load(userProfileProperties);

clientContext.executeQueryAsync(Function.createDelegate(this, function (sender, args) {

var firstname = userProfileProperties.get\_userProfileProperties()['FirstName'];

var name = userProfileProperties.get\_userProfileProperties()['PreferredName'];

var title = userProfileProperties.get\_userProfileProperties()['Title'];

var aboutMe = userProfileProperties.get\_userProfileProperties()['AboutMe'];

var picture = userProfileProperties.get\_userProfileProperties()['PictureURL'];

var html = "<div><h2>Welcome " + firstname + "</h2></div><div><div style='float: left; margin-left:10px'><img style='float:left;margin-right:10px' src='" + picture + "' /><b>Name</b>: " + name + "<br /><b>Title</b>: " + title + "<br />" + aboutMe + "</div></div>";

document.getElementById('UserProfileAboutMe').innerHTML = html;

}), Function.createDelegate(this, function (sender, args) {

console.log('The following error has occurred while loading user profile property: ' + args.get\_message());

}));

}, 'SP.UserProfiles.js');

}

## [Provisioning publishing features](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Provisioning.PublishingFeatures)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to perform common tasks with publishing sites that are hosted on Office 365:   * Provision and use page layouts, master pages and themes * Embed JavaScript in page layouts * Apply filters that control what site templates are available for subsites and what page layouts are available on the host web | This sample shows how to modify the UI of a page on the host site by deploying an app script part and adding it to a SharePoint page from the web part gallery.  **Typical scenarios**:   * Provision UI elements on publishing sites * Deploy custom master pages and themes to publishing sites * Control user options for using page layouts and site templates | The provider-hosted app uses CSOM to provision commonly used UI elements on publishing sites, and it uses JavaScript to create more dynamic experiences in page layouts that you can deploy to publishing sites. It also shows the differences between using master pages and themes in publishing sites. |

**Related samples**:

* [Branding.UIElementPersonalization](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.UIElementPersonalization)
* [Branding.ApplyBranding](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.ApplyBranding)
* [Core.ModifyPages](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.ModifyPages)
* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)

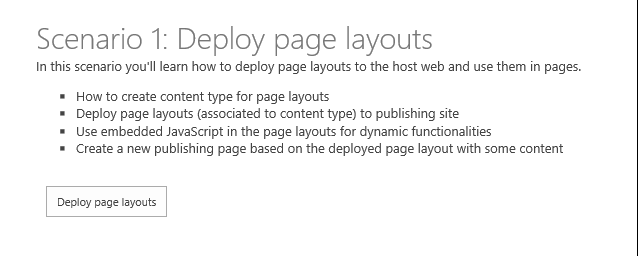
**Important note**:

In order to make all of the functionality in this sample work, you’ll need to activate the publishing features on your site. See [Enable publishing features](http://office.microsoft.com/en-us/sharepoint-server-help/enable-publishing-features-HA102802294.aspx) for instructions on how to do this.

The launch page presents you with three scenarios for customizing the UI of publishing sites: deploying page layouts, deploying master pages and themes, and filtering the available page layouts and site templates on the host site.

**Scenario 1 (deploying pages)** combines techniques we’ve already described in the [Page manipulation](#_Page_manipulation) and [Personalized UI elements](#_Personalized_UI_elements) sections. It also demonstrates how to deploy a custom page layout. The **Deploy page layouts** button in Figure 20 creates a new page layout and a page that uses that layout.

**Figure 20. Scenario 1 button for deploying page layouts**



You can see the new page by going to your host site and navigating to the newly created **demo** page (Figure 21). The page contains embedded JavaScript that displays user profile information.

**Figure 21. Page using new page layout that has personalized UI**



The sample uses the same technique described in the [Personalized UI elements](#_Personalized_UI_elements) section to display this users information on the page. It uses the same technique for adding a page that is described in the [Page manipulation](#_Page_manipulation_(Core.ModifyPages) section, although in this case it uses a PublishingPageInformation object to create the new page.

The sample adds a new page layout by uploading a file to the master page gallery and assigning it the page layout content type. The following code takes the path to a \*.aspx file (which you can deploy as a resource in your Visual Studio 2013 project) and adds it as a page layout in the master page gallery.

// Get the path to the file which we are about to deploy

List masterPageGallery = web.GetCatalog((int)ListTemplateType.MasterPageCatalog);

Folder rootFolder = masterPageGallery.RootFolder;

web.Context.Load(masterPageGallery);

web.Context.Load(rootFolder);

web.Context.ExecuteQuery();

var fileBytes = System.IO.File.ReadAllBytes(sourceFilePath);

// Use CSOM to upload the file in

FileCreationInformation newFile = new FileCreationInformation();

newFile.Content = fileBytes;

newFile.Url = UrlUtility.Combine(rootFolder.ServerRelativeUrl, fileName);

newFile.Overwrite = true;

Microsoft.SharePoint.Client.File uploadFile = rootFolder.Files.Add(newFile);

web.Context.Load(uploadFile);

web.Context.ExecuteQuery();

// Check out the file if needed

if (masterPageGallery.ForceCheckout || masterPageGallery.EnableVersioning)

{

if (uploadFile.CheckOutType == CheckOutType.None)

{

uploadFile.CheckOut();

}

}

// Get content type for ID to assign associated content type information

ContentType associatedCt = web.GetContentTypeById(associatedContentTypeID);

var listItem = uploadFile.ListItemAllFields;

listItem["Title"] = title;

listItem["MasterPageDescription"] = description;

// set the item as page layout

listItem["ContentTypeId"] = Constants.PAGE\_LAYOUT\_CONTENT\_TYPE;

// Set the associated content type ID property

listItem["PublishingAssociatedContentType"] = string.Format(";#{0};#{1};#", associatedCt.Name, associatedCt.Id);

listItem["UIVersion"] = Convert.ToString(15);

listItem.Update();

// Check in the page layout if needed

if (masterPageGallery.ForceCheckout || masterPageGallery.EnableVersioning)

{

uploadFile.CheckIn(string.Empty, CheckinType.MajorCheckIn);

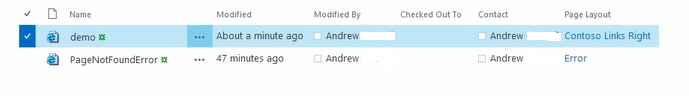
listItem.File.Publish(string.Empty);

}

web.Context.ExecuteQuery();

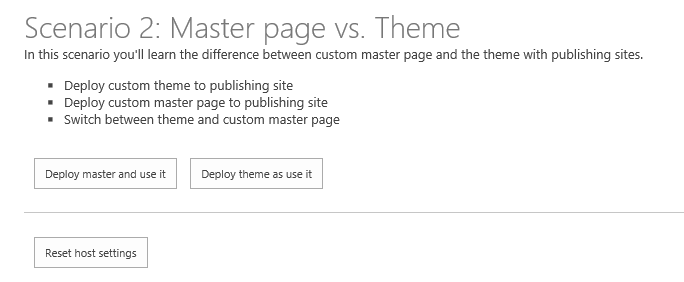
You can verify that your new page is using the newly created page layout by navigating to the **Pages** library (which you can reach from the **Site Contents** page) of the host site (Figure 22).

**Figure 22. New demo page in the Pages library**



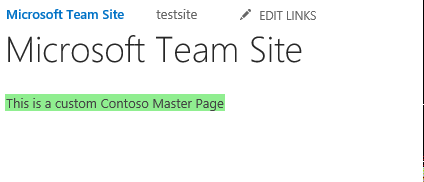
**Scenario 2 (deploying master pages and themes)** shows you how to deploy and set master pages and themes for the host site from a provider-hosted app (Figure 23).

**Figure 23. Deploying a master page and a theme**



When you click on the **Deploy master and use it** button, the sample deploys and applies a custom master page to the host site. You can see the newly deployed master page by visiting the home page of the site (Figure 24).

**Figure 24. Custom master page on the host site**



The sample adds a new master page by uploading a \*.master file to the master page gallery and assigning it the master page content type. The following code takes the path to a \*.master file (which you can deploy as a resource in your Visual Studio 2013 project) and adds it as a master page in the master page gallery.

string fileName = Path.GetFileName(sourceFilePath);

// Get the path to the file which we are about to deploy

List masterPageGallery = web.GetCatalog((int)ListTemplateType.MasterPageCatalog);

Folder rootFolder = masterPageGallery.RootFolder;

web.Context.Load(masterPageGallery);

web.Context.Load(rootFolder);

web.Context.ExecuteQuery();

// Get the file name from the provided path

var fileBytes = System.IO.File.ReadAllBytes(sourceFilePath);

// Use CSOM to upload the file in

FileCreationInformation newFile = new FileCreationInformation();

newFile.Content = fileBytes;

newFile.Url = UrlUtility.Combine(rootFolder.ServerRelativeUrl, fileName);

newFile.Overwrite = true;

Microsoft.SharePoint.Client.File uploadFile = rootFolder.Files.Add(newFile);

web.Context.Load(uploadFile);

web.Context.ExecuteQuery();

var listItem = uploadFile.ListItemAllFields;

if (masterPageGallery.ForceCheckout || masterPageGallery.EnableVersioning)

{

if (uploadFile.CheckOutType == CheckOutType.None)

{

uploadFile.CheckOut();

}

}

listItem["Title"] = title;

listItem["MasterPageDescription"] = description;

// Set content type as master page

listItem["ContentTypeId"] = Constants.MASTERPAGE\_CONTENT\_TYPE;

listItem["UIVersion"] = uiVersion;

listItem.Update();

if (masterPageGallery.ForceCheckout || masterPageGallery.EnableVersioning)

{

uploadFile.CheckIn(string.Empty, CheckinType.MajorCheckIn);

listItem.File.Publish(string.Empty);

}

web.Context.Load(listItem);

web.Context.ExecuteQuery();

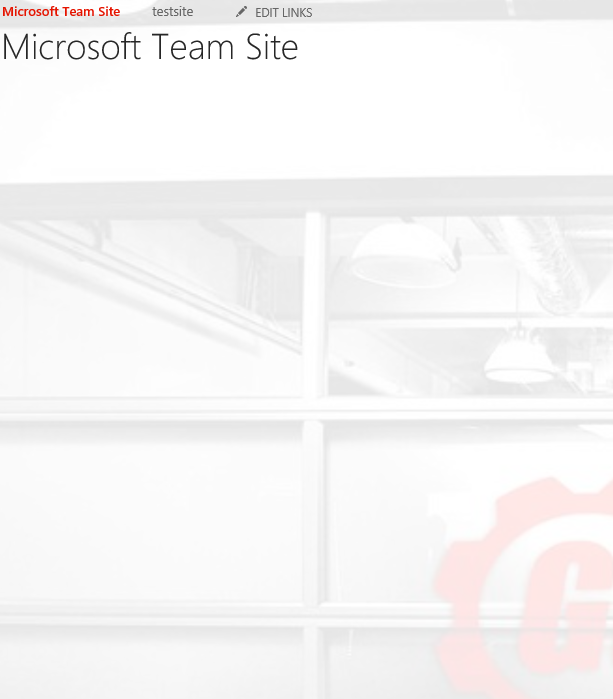
The next step is to set the URL of the newly deployed master page as the value for both the MasterUrl and CustomMasterUrl properties of the Web object that represents the site. The sample handles this with a single method that fetches the URL of the new master page in the master page gallery and then assigns that value to the Web.MasterUrl and Web.CustomMasterUrl properties.

// Assign master page to the host web

clientContext.Web.SetMasterPagesForSiteByName("contoso.master", "contoso.master");

When you click on the **Deploy theme and use it** button, the sample deploys and applies a custom theme to the host site. You can see the newly deployed theme by visiting the home page of the site (Figure 25).

**Figure 25. Custom theme on the host site**



The sample sets the color palette, background image, and font scheme of the theme by adding a new theme with those values (which you can deploy as resources inside your Visual Studio 2013 project) to the theme gallery. The following code creates the new theme.

List themesOverviewList = web.GetCatalog((int)ListTemplateType.DesignCatalog);

web.Context.Load(themesOverviewList);

web.Context.ExecuteQuery();

ListItemCreationInformation itemInfo = new ListItemCreationInformation();

Microsoft.SharePoint.Client.ListItem item = themesOverviewList.AddItem(itemInfo);

item["Name"] = themeName;

item["Title"] = themeName;

if (!string.IsNullOrEmpty(colorFileName))

{

item["ThemeUrl"] = UrlUtility.Combine(rootWeb.ServerRelativeUrl, string.Format(Constants.THEMES\_DIRECTORY, Path.GetFileName(colorFileName)));

}

if (!string.IsNullOrEmpty(fontFileName))

{

item["FontSchemeUrl"] = UrlUtility.Combine(rootWeb.ServerRelativeUrl, string.Format(Constants.THEMES\_DIRECTORY, Path.GetFileName(fontFileName)));

}

if (!string.IsNullOrEmpty(backgroundName))

{

item["ImageUrl"] = UrlUtility.Combine(rootWeb.ServerRelativeUrl, string.Format(Constants.THEMES\_DIRECTORY, Path.GetFileName(backgroundName)));

}

item["DisplayOrder"] = 11;

item.Update();

web.Context.ExecuteQuery();

The next step is to set this new theme as the theme for the site. The following code does this by fetching the theme from the theme gallery and then applying its values to the host site.

CamlQuery query = new CamlQuery();

// Find the theme by themeName

string camlString = string.Format(CAML\_QUERY\_FIND\_BY\_FILENAME, themeName);

query.ViewXml = camlString;

var found = themeList.GetItems(query);

rootWeb.Context.Load(found);

LoggingUtility.Internal.TraceVerbose("Getting theme: {0}", themeName);

rootWeb.Context.ExecuteQuery();

if (found.Count > 0)

{

ListItem themeEntry = found[0];

//Set the properties for applying custom theme which was just uploaded

string spColorURL = null;

if (themeEntry["ThemeUrl"] != null && themeEntry["ThemeUrl"].ToString().Length > 0)

{

spColorURL = UrlUtility.MakeRelativeUrl((themeEntry["ThemeUrl"] as FieldUrlValue).Url);

}

string spFontURL = null;

if (themeEntry["FontSchemeUrl"] != null && themeEntry["FontSchemeUrl"].ToString().Length > 0)

{

spFontURL = UrlUtility.MakeRelativeUrl((themeEntry["FontSchemeUrl"] as FieldUrlValue).Url);

}

string backGroundImage = null;

if (themeEntry["ImageUrl"] != null && themeEntry["ImageUrl"].ToString().Length > 0)

{

backGroundImage = UrlUtility.MakeRelativeUrl((themeEntry["ImageUrl"] as FieldUrlValue).Url);

}

// Set theme for demonstration

// TODO: Why is shareGenerated false? If deploying to root an inheriting, then maybe use shareGenerated = true.

web.ApplyTheme(spColorURL,

spFontURL,

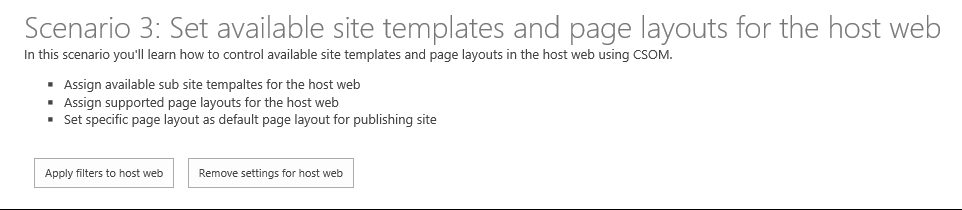
backGroundImage,

false);

web.Context.ExecuteQuery();

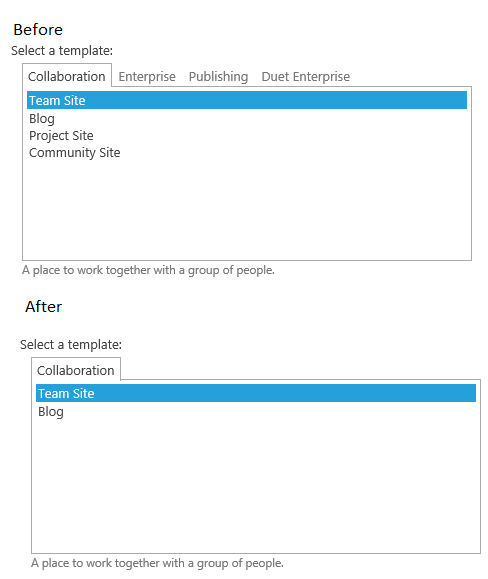
**Scenario 3 (filtering available page layouts and site templates)** shows you how to limit the options that users have when applying templates to new sites and layouts to new pages (Figure 26).

**Figure 26. Filter available page layouts and site templates**



When you click on the **Apply filters to host web** button, the sample sets a custom page layout as the default and one additional page layout as the only other option for any new pages that a user creates. The sample also reduces the number of available options for users when they create new subsites. Figure 26 shows what the site template selection box looks like both before and after the filters have been applied.

**Figure 26. Site template selection before and after sample filters have been applied**



The sample sets both the default and available page layouts by passing the associated \*.aspx files to methods to extension methods.

List<string> pageLayouts = new List<string>();

pageLayouts.Add("ContosoLinksBelow.aspx");

pageLayouts.Add("ContosoLinksRight.aspx");

clientContext.Web.SetAvailablePageLayouts(clientContext.Web, pageLayouts);

// Set default page layout for the site

clientContext.Web.SetDefaultPageLayoutForSite(clientContext.Web, "ContosoLinksBelow.aspx");

The sample sets the available site templates by doing something similar. In this case it passes the WebTemplateEntity objects that define each site template to an extension method called SetAvailableWebTemplates.

List<WebTemplateEntity> templates = new List<WebTemplateEntity>();

templates.Add(new WebTemplateEntity() { LanguageCode = "1035", TemplateName = "STS#0" });

templates.Add(new WebTemplateEntity() { LanguageCode = "", TemplateName = "STS#0" });

templates.Add(new WebTemplateEntity() { LanguageCode = "", TemplateName = "BLOG#0" });

clientContext.Web.SetAvailableWebTemplates(templates);

(Note: The 1035 language code maps to the Finnish language, which isn’t available on the site from which the screen shots were taken.)

All three of these extension methods – SetAvailablePageLayouts, SetDefaultPageLayoutForSite, and SetAvailableWebTemplates work in the same way. They create XML documents containing key/value pairs that define the available and default layouts and the available templates. They then pass these documents to an additional extension method called SetPropertyBagValue. This method is implemented in the [OfficeDevPnPCore extension component](https://github.com/OfficeDev/PnP/tree/dev/OfficeDevPnP.Core), and once it sets up the appropriate property bags, these property bags are then used to filter options in the interface.

The SetAvailableWebTemplates method illustrates the pattern most fully.

public static void SetAvailableWebTemplates(this Web web, List<WebTemplateEntity> availableTemplates)

{

string propertyValue = string.Empty;

LanguageTemplateHash languages = new LanguageTemplateHash();

foreach (var item in availableTemplates)

{

AddTemplateToCollection(languages, item);

}

if (availableTemplates.Count > 0)

{

XmlDocument xd = new XmlDocument();

XmlNode xmlNode = xd.CreateElement("webtemplates");

xd.AppendChild(xmlNode);

foreach (var language in languages)

{

XmlNode xmlLcidNode = xmlNode.AppendChild(xd.CreateElement("lcid"));

XmlAttribute xmlAttribute = xd.CreateAttribute("id");

xmlAttribute.Value = language.Key;

xmlLcidNode.Attributes.SetNamedItem(xmlAttribute);

foreach (string item in language.Value)

{

XmlNode xmlWTNode = xmlLcidNode.AppendChild(xd.CreateElement("webtemplate"));

XmlAttribute xmlAttributeName = xd.CreateAttribute("name");

xmlAttributeName.Value = item;

xmlWTNode.Attributes.SetNamedItem(xmlAttributeName);

}

}

propertyValue = xmlNode.OuterXml;

}

//Save the xml entry to property bag

web.SetPropertyBagValue(AvailableWebTemplates, propertyValue);

//Set that templates are not inherited

web.SetPropertyBagValue(InheritWebTemplates, "False");

}

The InheritWebTemplates property bag makes sure that any templates that would normally be inherited from the parent site will also be ignored when creating subsites.

# Creating UX controls with provider-hosted apps

The samples in this section demonstrate how to create UX controls in provider-hosted apps that work like UX controls on the host web. These samples use JavaScript and the SharePoint JSOM to talk to SharePoint. Like the sample described in the [showing apps and data in dialogs](#_Showing_apps_and) section, these samples use the [cross-domain library](http://msdn.microsoft.com/en-us/library/office/fp179927(v=office.15).aspx) to handle the cross-domain calls from the provider-hosted app to the host site domain.

## [People Picker](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.PeoplePicker)

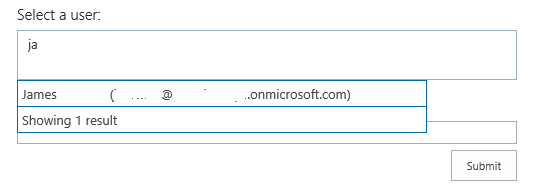
|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to implement a people picker control in a provider-hosted app. This control works much like the people picker control on a SharePoint host site. When the user starts typing a name into the text input box, the control searches the user profile store for matches, and potential matches display in the UI. | This control is helpful in cases where you need to add site collection administrators to sites, users to new sites, users to groups, etc. This would be especially helpful in apps that use the remote site provisioning pattern. You could, for example, add people picker capability to an on-demand remote site provisioning app. | The app creates an HTML page that conforms to the JSOM people picker requirements, and then adds and configures the control. It uses the ClientPeoplePickerWebServiceInterface object in the JSOM to query the user profile store. |

**Related samples**:

* [Core.JavaScriptCustomization](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.JavaScriptCustomization)
* [Provisioning.OnPrem.Async](https://github.com/OfficeDev/PnP/tree/dev/Samples/Provisioning.OnPrem.Async)

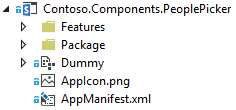
This provider-hosted app displays a simple configurable and extensible people picker control that runs on a remote host and queries the user profile store on the host site (Figure 27).

**Figure 27. People picker control**



The app for SharePoint project in this sample contains a module named **Dummy** to ensure that when the app deploys it will create an app web. An app web is required for [cross-domain calls](http://msdn.microsoft.com/en-us/library/office/fp179927(v=office.15).aspx) from JavaScript running on a remote site (Figure 28).

**Figure 28. Visual Studio 2013: Dummy module causes an app web to be created**

[](https://camo.githubusercontent.com/1091208a45064d9c8362f495432c62ec7b1b0321/687474703a2f2f692e696d6775722e636f6d2f4555445872766f2e706e67)

The Scripts folder of the Core.PeoplePickerWeb project contains app.js and peoplepickercontrol.js files (along with people picker resource files for additional language support). The app.js file fetches client context by using the cross-domain library and hooks the HTML in the Default.aspx file into the people picker control. The Default.aspx file contains the div tags that implement both the text box and the people search capability.

<div id="divAdministrators" class="cam-peoplepicker-userlookup ms-fullWidth">

<span id="spanAdministrators"></span>

<asp:TextBox ID="inputAdministrators" runat="server" CssClass="cam-peoplepicker-edit" Width="70"></asp:TextBox>

</div>

<div id="divAdministratorsSearch" class="cam-peoplepicker-usersearch ms-emphasisBorder"></div>

<asp:HiddenField ID="hdnAdministrators" runat="server" />

The app.js file then creates and configures a people picker.

//Make a people picker control

//1. context = SharePoint Client Context object

//2. $('#spanAdministrators') = SPAN that will 'host' the people picker control

//3. $('#inputAdministrators') = INPUT that will be used to capture user input

//4. $('#divAdministratorsSearch') = DIV that will show the 'dropdown' of the picker

//5. $('#hdnAdministrators') = INPUT hidden control that will host a resolved users

peoplePicker = new CAMControl.PeoplePicker(context, $('#spanAdministrators'), $('#inputAdministrators'), $('#divAdministratorsSearch'), $('#hdnAdministrators'));

// required to pass the variable name here!

peoplePicker.InstanceName = "peoplePicker";

// Hookup everything

peoplePicker.Initialize();

The people picker control queries the ClientPeoplePickerWebServiceInterface object in the JSOM to initiate searches for users who match the character strings entered by the user.

if (searchText.length >= parent.GetMinimalCharactersBeforeSearching()) {

resultDisplay = 'Searching...';

if (typeof resultsSearching != 'undefined') {

resultDisplay = resultsSearching;

}

var searchbusy = parent.Format('<div class=\'ms-emphasisBorder\' style=\'width: 400px; padding: 4px; border-left: none; border-bottom: none; border-right: none; cursor: default;\'>{0}</div>', resultDisplay);

parent.PeoplePickerDisplay.html(searchbusy);

//display the suggestion box

parent.ShowSelectionBox();

var query = new SP.UI.ApplicationPages.ClientPeoplePickerQueryParameters();

query.set\_allowMultipleEntities(false);

query.set\_maximumEntitySuggestions(2000);

query.set\_principalType(parent.GetPrincipalType());

query.set\_principalSource(15);

query.set\_queryString(searchText);

var searchResult = SP.UI.ApplicationPages.ClientPeoplePickerWebServiceInterface.clientPeoplePickerSearchUser(parent.SharePointContext, query);

// update the global queryID variable so that we can correlate incoming delegate calls later on

parent.\_queryID = parent.\_queryID + 1;

var queryIDToPass = parent.\_queryID;

parent.\_lastQueryID = queryIDToPass;

// make the SharePoint request

parent.SharePointContext.executeQueryAsync(Function.createDelegate(this, function () { parent.QuerySuccess(queryIDToPass, searchResult); }),

Function.createDelegate(this, function () { parent.QueryFailure(queryIDToPass); }));

The [Core.PeoplePicker](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.PeoplePicker) repository on GitHub contains additional instructions for configuring and extending this control.

## [Taxonomy menu](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.TaxonomyMenu)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to implement a localizable taxonomy menu control that is populated from the term store in a provider-hosted app. The app also sets up the required term store languages, groups, sets, and terms for populating the menu. It checks the user’s language preference to determine the display language. | This control is helpful in cases where you need to create term store capabilities that run remotely. This could be especially helpful when you are using the remote site provisioning pattern and want to set up a new site collection with a term store and display it in the user interface of the host site. | The app implements a TaxonomyHelper class (CSOM) that sets up the term store and populates it with terms. It then uploads into the site’s root folder a JavaScript file that displays the navigational links. |

**Related samples**:

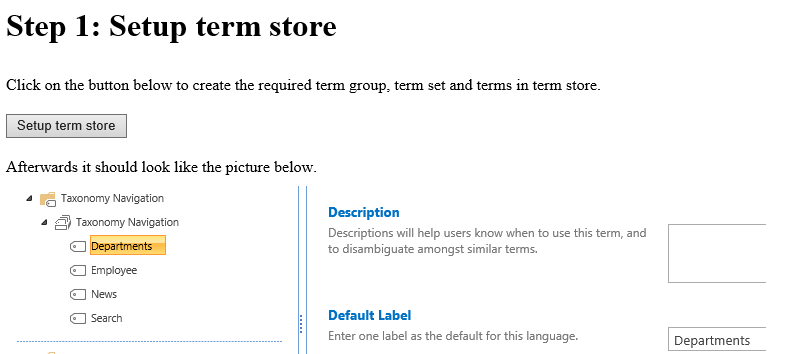
* [Core.JavaScriptCustomization](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.JavaScriptCustomization)
* [Provisioning.OnPrem.Async](https://github.com/OfficeDev/PnP/tree/dev/Samples/Provisioning.OnPrem.Async)

**Alternative approaches**:

* [Branding.CustomCSS](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.CustomCSS) (CSS injection to the host web)
* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)

The launch page of the app presents you with two steps. The first step sets up the term store on the host site. It uses CSOM objects and methods to create a term group and set, and then populates the term set with four terms: Departments, Employee, News, and Search (Figure 29).

**Figure 29. User interface for setting up the term store**



When you choose the Setup term store button, the app does two things: makes sure that the required languages (English, French, German, and Swedish) are enabled in the term store and then creates a term group and term set and populates the term set with the four new terms.

The following code in the TaxonomyHelper class checks to see if the required languages are enabled, and if not, it enables them.

var languages = new int[] { 1031, 1033, 1036, 1053 };

Array.ForEach(languages, l => {

if (!termStore.Languages.Contains(l))

termStore.AddLanguage(l);

});

termStore.CommitAll();

clientContext.ExecuteQuery();

The following code in the TaxonomyHelper class creates the term group.

termGroup = termStore.CreateGroup("Taxonomy Navigation", groupId);

clientContext.Load(termGroup);

clientContext.ExecuteQuery();

The following code in the TaxonomyHelper class creates the term set.

termSet = termGroup.CreateTermSet("Taxonomy Navigation", termSetId, 1033);

termSet.SetCustomProperty("\_Sys\_Nav\_IsNavigationTermSet", "True");

clientContext.Load(termSet, ts => ts.Terms);

clientContext.ExecuteQuery();

Finally, the following code in the same TaxonomyHelper class creates each new term, along with labels for the French, German, and Swedish languages. It also sets a value for the \_Sys\_Nav\_SimpleLinkUrl property, which is equivalent to the **Simple Link or Header** property that you can find in the Term Store Management Tool. You can see that in this case the URL for each term simply points back to the root site.

var term = termSet.CreateTerm(termName, 1033, Guid.NewGuid());

term.CreateLabel(termNameGerman, 1031, false);

term.CreateLabel(termNameFrench, 1036, false);

term.CreateLabel(termNameSwedish, 1053, false);

term.SetLocalCustomProperty("\_Sys\_Nav\_SimpleLinkUrl", clientContext.Web.ServerRelativeUrl);

The second step inserts the topnav.js file into the root folder of the host site. This file contains the JavaScript that inserts the links from this term set into the navigation of the host site’s home page. The app’s UI also shows you how the navigational links will appear on the host site after the app has uploaded the JavaScript file (Figure 30).

**Figure 30. User interface for uploading the taxonomy menu JavaScript file**



The following code in the topnav.js file uses the JSOM to check for the user’s preferred language.

var targetUser = "i:0#.f|membership|" + \_spPageContextInfo.userLoginName;

context = new SP.ClientContext.get\_current();

var peopleManager = new SP.UserProfiles.PeopleManager(context);

var userProperty = peopleManager.getUserProfilePropertyFor(targetUser, "SPS-MUILanguages");

It then checks to see if the user’s language preference matches one of the enabled languages. If it finds a match, the following code gets the terms and the associated labels for the user’s preferred language.

while (termEnumerator.moveNext()) {

var currentTerm = termEnumerator.get\_current();

var label = currentTerm.getDefaultLabel(lcid);

termItems.push(currentTerm);

termLabels.push(label);

context.load(currentTerm);

}

Finally, the following code in the topnav.js file inserts links containing the terms into the top navigational element of the host site.

html += "<ul style='margin-top: 0px; margin-bottom: 0px;'>"

for (var i in termItems) {

var term = termItems[i];

var termLabel = termLabels[i];

var linkName = termLabel.get\_value() != 0 ? termLabel.get\_value() : term.get\_name();

var linkUrl = term.get\_localCustomProperties()['\_Sys\_Nav\_SimpleLinkUrl'];

html += "<li style='display: inline;list-style-type: none; padding-right: 20px;'><a href='" + linkUrl + "'>" + linkName + "</a></li>";

}

html += "</ul>";

$('#DeltaTopNavigation').html(html);

SP.UI.Notify.removeNotification(nid);

## [Taxonomy picker](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.TaxonomyPicker)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to implement a taxonomy picker control in a provider-hosted app. When the user starts typing a term into the text input box, the control searches the term store for potential matches and displays them in a list underneath the input box. | This control is helpful in cases where you need to add terms to the host site’s taxonomy or use terms from the host site’s taxonomy in the context of a remotely hosted app. | The app creates an HTML page that conforms to the JSOM taxonomy picker requirements, and then adds and configures the control. It uses the JSOM to query the host site’s term store. |

**Related samples**:

* [Core.JavaScriptCustomization](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.JavaScriptCustomization)

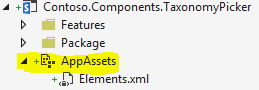
This provider-hosted app displays a simple configurable and extensible taxonomy picker control that runs on a remote host and queries the term store on the host site (Figure 31).

**Figure 31. Taxonomy picker control**



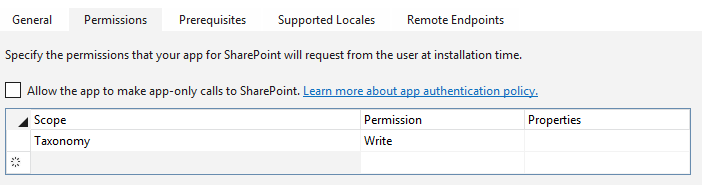
The app for SharePoint project in this sample contains a module named **AppAssets** to ensure that when the app deploys it will create an app web. An app web is required for [cross-domain calls](http://msdn.microsoft.com/en-us/library/office/fp179927(v=office.15).aspx) from JavaScript running on a remote site (Figure 32).

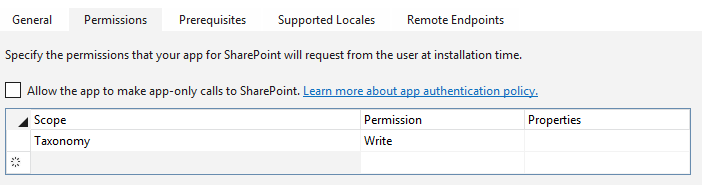
**Figure 32. Visual Studio 2013: AppAssets module causes an app web to be created**



The Taxonomy Picker communicates with SharePoint’s Managed Metadata Service, which requires special permissions in the app model. The app must have write permission at the Taxonomy permission scope so that it can read from closed term sets and write to open term sets. Make sure that the AppManifest.xml file has set this permission at the appropriate scope (Figure 33).

**Figure 33. Visual Studio 2013: Give the app write permission at the Taxonomy scope**





The Scripts folder of the Core.TaxonomyPickerWeb project contains app.js and taxonomypickercontrol.js files (along with a taxonomy picker resource file for additional language support). The app.js file fetches client context by using the cross-domain library and hooks the HTML in the Default.aspx file into the taxonomy picker control. The Default.aspx file contains the hidden field that implements both the text box and the taxonomy picker capability. It also adds a bulleted list for displaying suggestions returned form the term store.

<div style="left: 50%; width: 600px; margin-left: -300px; position: absolute;">

<table>

<tr>

<td class="ms-formlabel" valign="top"><h3 class="ms-standardheader">Keywords Termset:</h3></td>

<td class="ms-formbody" valign="top">

<div class="ms-core-form-line" style="margin-bottom: 0px;">

<asp:HiddenField runat="server" id="taxPickerKeywords" />

</div>

</td>

</tr>

</table>

<asp:Button runat="server" OnClick="SubmitButton\_Click" Text="Submit" />

<asp:BulletedList runat="server" ID="SelectedValues" DataTextField="Label" />

</div>

The app.js file then creates and configures a taxonomy picker.

//load scripts for calling taxonomy APIs

$.getScript(layoutsRoot + 'init.js',

function () {

$.getScript(layoutsRoot + 'sp.taxonomy.js',

function () {

//binf the taxonomy picker to the default keywords termset

$('#taxPickerKeywords').taxpicker({ isMulti: true, allowFillIn: true, useKeywords: true }, context);

});

});

The taxonomy picker control uses the following code to open a TaxonomySession in the JSOM to load all of the terms from the term store.

//Get the taxonomy session using CSOM

var taxSession = SP.Taxonomy.TaxonomySession.getTaxonomySession(spContext);

//Use the default term store...this could be extended here to support additional term stores

var termStore = taxSession.getDefaultSiteCollectionTermStore();

//get the termset based on the properties of the termset

if (this.Id != null)

this.RawTermSet = termStore.getTermSet(this.Id); //get termset by id

else if (this.UseHashtags)

this.RawTermSet = termStore.get\_hashTagsTermSet(); //get the hashtags termset

else if (this.UseKeywords)

this.RawTermSet = termStore.get\_keywordsTermSet(); //get the keywords termset

//get ALL terms for the termset and we will organize them in the async callback

this.RawTerms = this.RawTermSet.getAllTerms();

spContext.load(this.RawTermSet);

spContext.load(this.RawTerms);

spContext.executeQueryAsync(Function.createDelegate(this, this.termsLoadedSuccess), Function.createDelegate(this, this.termsLoadedFailed));

It then looks for potential matches from the loaded terms, and adds new terms to the term store.

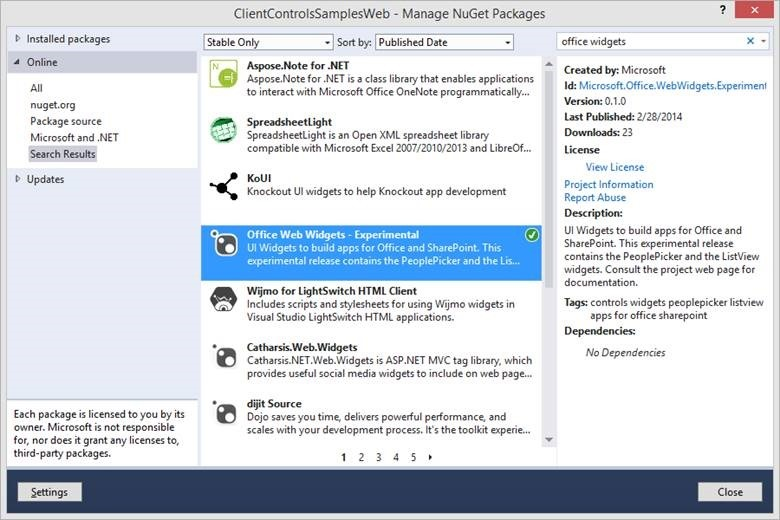
The [Core.TaxonomyPicker](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.TaxonomyPicker) repository on GitHub contains additional instructions for configuring and extending this control.

# Using OfficeWebWidgets

The [Office Web Widgets library](http://www.nuget.org/packages/Microsoft.Office.WebWidgets.Experimental/), which has been made available as a NuGet package, is an experimental (and therefore not entirely complete) set of controls that display information from a SharePoint host site in the context of a provider-hosted app. The controls are designed to look and act as much like SharePoint-hosted controls as possible. They have some limitations and bugs in their current form, and for that reason their primary goal is to prompt feedback from the developer community. The [Office Web Widgets – Experimental](http://blogs.msdn.com/b/officeapps/archive/2014/03/07/office-web-widgets-experimental.aspx) blog post on MSDN discusses the limitations and bugs more fully, and also contains detailed instructions about how to implement them. The [Office Web Widgets – Experimental overview](http://msdn.microsoft.com/en-us/library/office/dn636913(v=office.15).aspx) article on MSDN also describes in detail how to implement these controls, and the [Office Web Widgets – Experimental Demo sample](http://code.msdn.microsoft.com/office/SharePoint-2013-Office-Web-6d44aa9e#content) on Code Gallery contains another sample implementation of this library.

The [Core.OfficeWebWidgets](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.OfficeWebWidgets) sample will load the OfficeNuGet package automatically. You can add this library to your own Visual Studio 2013 projects by right-clicking the name of your project, selecting **Manage NuGet Packages** and searching for Office Widgets. You’ll see the package in the center column (Figure 34).

**Figure 34. Visual Studio 2013: Adding the Office Web Widgets package to your web project**



The package will add an OfficeControls.css file and a number of Office.Controls JavaScript files to the Scripts folder of your web project.

## [OfficeWebWidgets](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.OfficeWebWidgets)

|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to implement the people picker and list view controls that are included in the Office Web Widgets library. | These controls demonstrate that you can provide a fully featured user experience for interacting with SharePoint data in the context of a provider-hosted app. | The app inserts the controls in the HTML of the launch page and then configures and creates them in the app.js JavaScript file that is included in the Scripts folder of the web project. |

**Related samples**:

* [Core.JavaScriptCustomization](https://github.com/OfficeDev/PnP/tree/dev/Scenarios/Core.JavaScriptCustomization)
* [Core.PeoplePicker](https://github.com/OfficeDev/PnP/tree/dev/Components/Core.PeoplePicker)

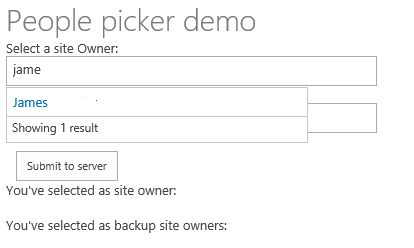
**Alternative approaches**:

* [Branding.CustomCSS](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.CustomCSS) (CSS injection to the host web)
* [Core.JavaScriptInjection](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjection) (JavaScript injection)
* [Core.JavaScriptInjectionJSOM](https://github.com/OfficeDev/PnP/tree/dev/Samples/Core.JavaScriptInjectionJSOM) (JavaScript injection with JSOM)

The Core.OfficeWebWidgets sample demonstrates two simple implementations of the people picker and list view controls that are included in the Office Web Widgets library.

The launch page displays the people picker control, which behaves much like the sample described in the [People Picker](#_People_Picker) section. This version allows you to add multiple users, and it contains two text boxes for user input (Figure 35).

**Figure 35. People picker demo from the Office Web Widgets sample**



The following code from the app.js file creates an instance of the people picker control.

//Create the people picker

var siteOwnerPeoplePicker = new Office.Controls.PeoplePicker(document.getElementById("peoplePickerSiteOwner"),

{

allowMultipleSelections: false,

placeholder: "Please choose an site owner",

onChange: handleSiteOwnerChange

});

The following code from the Default.aspx file of the web project includes the people picker control inside a <div> tag.

<div id="peoplePickerBackupSiteOwners" data-office-control="Office.Controls.PeoplePicker" data-office-options='{ "placeholder" : "Please choose one or more backup site owner",

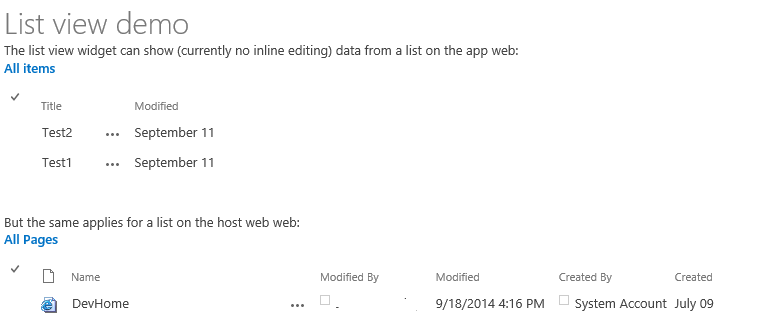
"allowMultipleSelections" : true,

"onChange" : handleSiteOwnerBackupChange

}'></div>

The launch page also displays the list view control, which can display lists from both the host web and the app web of the SharePoint site (Figure 36).

**Figure 36. List view demo from the Office Web Widgets sample**



The following code from the Default.aspx file includes two <div> tags for displaying the two instances of the list view control.

The list view widget can show (currently no inline editing) data from a list on the app web:<br />

<div id="listViewAppWeb"></div>

<br />

<br />

But the same applies for a list on the host web web:<br />

<div id="listViewHostWeb"></div>

The following code from the app.js file creates the two instances of the list view control.

var listViewAppWeb = new Office.Controls.ListView(document.getElementById("listViewAppWeb"),

{

listUrl: appWebUrl + "/\_api/web/lists/getbytitle('Announcements')"

});

var listViewHostWeb = new Office.Controls.ListView(document.getElementById("listViewHostWeb"),

{

listUrl: spHostUrl + "/\_api/web/lists/getbytitle('Site Pages')"

});

# Better performance through caching

Two effective mechanisms for improving performance in provider-hosted apps by limiting the number of remote calls to SharePoint are HTTP cookies and HTML 5 local storage. The following sample demonstrates how to do both so that you can evaluate the advantages and limitations of both approaches in your own provider-hosted apps.

## [Performance caching](https://github.com/OfficeDev/PnP/tree/dev/Samples/Performance.Caching)

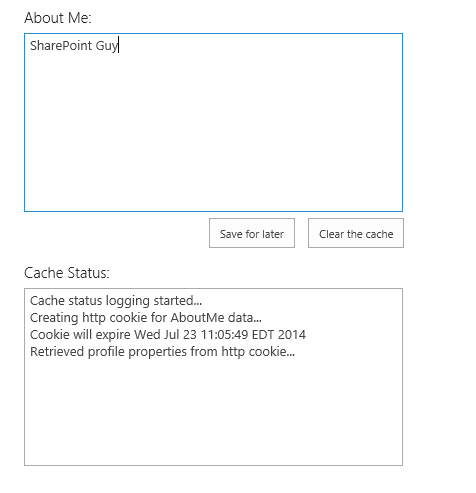
|  |  |  |
| --- | --- | --- |
| **What this demonstrates** | **Why you would want to use this**: | **How the app works** |
| How to improve performance in apps for SharePoint by using HTTP cookies and HTML 5 local storage. | You’ll often want to limit the number of calls that your app must make to SharePoint, because performance of both your app and the host site can degrade if you communicate with the host site more often than is absolutely necessary. You’ll also want to know how to use both HTTP cookies and HTML 5 local storage, since either approach might be a better match in light of the specific requirements of any given project. | The sample includes to SharePoint-hosted apps that allow you to view the **About Me** section of your user profile. You can add data to this section and save it for later. The app does not update user profile information but simply caches the information so that it can be used later. One sample uses HTTP cookies, and the other uses HTML 5 local storage. |

**Related samples**:

* [Branding.UIElementPersonalization](https://github.com/OfficeDev/PnP/tree/dev/Samples/Branding.UIElementPersonalization)

**HTTP cookies.** The launch page of the HTTP cookie demonstration displays information from the **About Me** section of your user profile and displays it in a text box. The second text box tells you whether a new cookie was created and when the existing cookie will expire (Figure 37). Note that the information stored in cookies can’t be larger than 4095 bytes.

**Figure 37. About Me data in the HTTP cookie caching sample**



The app.js file in the Scripts folder of the web project defines the behavior of the **Save for later** button. The code first checks to see that cookies are enabled in the browser by setting a test cookie. If cookies are enabled, it checks to see whether the user profile information is already stored in a cookie. If not, it uses the JSOM to look up the **About Me** information, store it in a cookie, and then display the information in the browser.

The following function sets the cookie and the expiration date.

function setCookie(key, value, expiry, path, domain, secure) {

var todaysDate = new Date();

todaysDate.setTime(todaysDate.getTime());

if (expiry == "") { expiry = "1"; }

// line below sets for n number of days - for hours, remove \* 24 - for minutes remove \* 60 \* 24

if (expiry) {

expiry = expiry \* 1000 \* 60 \* 60 \* 24;

}

var newExpiry = new Date(todaysDate.getTime() + (expiry));

document.cookie = key + "=" + escape(value) +

( ( expiry ) ? ";expires=" + newExpiry : "" ) +

( ( path ) ? ";path=" + path : "" ) +

( ( domain ) ? ";domain=" + domain : "" ) +

((secure) ? ";secure" : "");

cachingStatus += "\n" + "Creating http cookie for AboutMe data...";

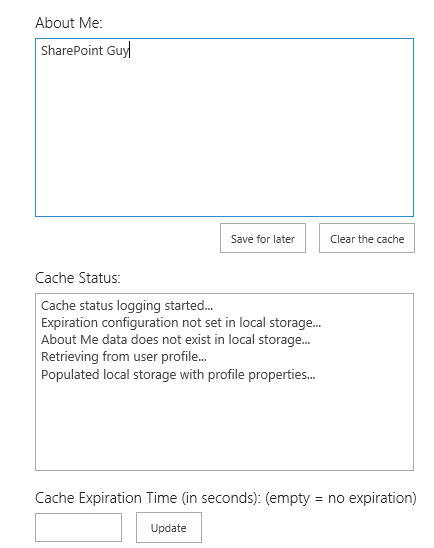
cachingStatus += "\n" + "Cookie will expire " + newExpiry;

$('#status').text(cachingStatus);

}

**HTML 5 local storage**. The launch page of the HTML 5 local storage demonstration also displays information from the **About Me** section of your user profile, along with information about the cached data in a second text box. It also displays the expiration time (if any) of the cached information (Figure 38).

**Figure 38. About Me data in the HTML 5 local storage sample**



The app.js file in the Scripts folder of the web project defines the behavior of the **Save for later** button. The app first checks to see that local storage is enabled with the following function.

isHtml5StorageSupported = function () {

try {

return 'localStorage' in window && window['localStorage'] !== null;

} catch (e) {

return false;

}

return false;

}

If local storage is supported, the checks to see whether the user profile information is already stored there. If not, it uses the JSOM to look up the **About Me** information, store it in a local storage, and then display the information in the browser. The following code stores the **About Me** information in a key named “aboutMeValue.”

var aboutMeValue = personProperties.get\_userProfileProperties()['AboutMe'];

$('#aboutMeText').val(aboutMeValue);

// add to local storage

localStorage.setItem("aboutMeValue", aboutMeValue);

setLocalStorageKeyExpiry("aboutMeValue");

cachingStatus += "\n" + "Populated local storage with profile properties...";

$('#status').val(cachingStatus);

The **Clear the cache** button removes that key, looks up the **About Me** information in your user profile, and creates a fresh local storage key to store that information.

The app doesn’t set an expiration time by default, but the app.js file does contain the following function, which sets an expiration time for the cached data.

function setLocalStorageKeyExpiry(key) {

// Check for expiration config values

var expiryConfig = localStorage.getItem(expiryConfigKey);

// Check for existing expiration stamp

var existingStamp = localStorage.getItem(key + expiryKeySuffix);

// Override cached setting if a user has entered a value that is different than what is stored

if (expiryConfig != null) {

var currentTime = Math.floor((new Date().getTime()) / 1000);

expiryConfig = parseInt(expiryConfig);

var newStamp = Math.floor((currentTime + expiryConfig));

localStorage.setItem(key + expiryKeySuffix, newStamp);

// Log status to window

cachingStatus += "\n" + "Setting expiration for the " + key + " key...";

$('#status').val(cachingStatus);

}

else {

}

}

Before looking for the information stored in the local storage key, it will check to see if the key as expired by using the same isKeyExpired function that is used in the sample described in the [Personalized UI elements](#_Personalized_UI_elements) section.