Kinds of Apps

Let's talk about different kinds of apps. Now, the way I understand anything is easy to, when there's a big complex topic, to say where do you start? Categorizing things within that big topic is a good way to start. So, you can categorize apps in various ways, but a very convenient way, or two different ways; either you can categorize them by their user interface type, or you can categorize them based on how they are hosted. So, when I say user interface type, remember that the app, no matter what kind of app it is, no matter what kind of app it is, it is running from a different URL. And by definition, if I was to open the browser, and if I just HTTP to that URL, I would get the full screen experience of that app. So every app supports the full screen experience. In fact, if you don't want the full screen experience, you actually have to put in extra work to prevent that full screen UI. Apps can also run as a "pretend I'm a webpart." As in users in SharePoint, they're very used to webparts. They love being able to put the page in a RIP mode, and drop a webpart, and do something useful with it. Apps can work like webparts. Now I said pretend a webpart, because a lot of things that we're used to in webparts are actually not available in apps. These webparts that run apps, are also called as app parts or client webparts. So, I'll use the term either app webpart or client webpart, to refer to an app running as a webpart. The app running as a webpart has got a lot of limitations. It's basically a glorified IFrame underneath the scenes, so I would basically say that it's pretending like it's a webpart. It's really not a webpart. And the third possible way that you can categorize apps from a UI perspective is when you launch them using a SharePoint custom action. If you watched my previous videos, you would know that custom actions are a mechanism for us to be able to create links at various places in SharePoint, like inside actions and the ECB menu, and site settings, and so on and so forth. So you could have a button in the ribbon, for instance, that launches an app. That's a perfectly good example of a UI experience for an app. So, how do you categorize them by UI type? Full screen takes over your entire browser, runs pretend I'm a webpart, which is a glorified IFrame, and running SharePoint as a custom action. So, running, launching the app through SharePoint is a custom action, so click on a link, and it launches the app. They can also be categorized based on how they are hosted. Sometimes apps can do whatever they need to using purely SharePoint hosted code. Using JavaScript code I mean. So, the app can be completely served out of SharePoint, and SharePoint only, because it is not running any code on the SharePoint server. It can run whatever it wants in JavaScript, but it's not running anything on the SharePoint server. No C# server side code, or VB.NET for that matter. So, that's called as a SharePoint hosted app. But sometimes you want server side code, and in that scenario, you have two subcategorizations. One is with you set up the server ahead of time. That's called as a provider-hosted app. And the other in which a server area is created for you any time the app is installed. If the app is installed 20 times, 20 server side instances are created for you. It's not as bad as it sounds, and 20 sounds like a pretty big, bad number, but the reality is they're all copies of each other, and actually when you uninstall the app, that server side area disappears, so it's actually no headache for us. It's actually a pretty good option. So that's called as an auto-hosted app. It applies only in Office 365 as of now, and when you basically install an app in Office 365, if it is an auto-hosted app, it will provision an area behind the scenes in Azure. We don't know that it's in Azure really. It's a black box to us. So it's not like we're going to see it in our Azure portal, and we'll be able to tweak it otherwise. No. It's a total black box for us. It basically creates an area in Azure for us, and the app runs over there merrily. So, the important thing to realize over here is that these two categorizations aren't exclusive to each other. So you can have a provider-hosted app that runs as a client webpart, or you can have an auto-hosted app that runs in full screen. But this is a good way for us to be able to understand categorizations of apps. So this is, again this is very important, it's a very important concept that we need to get in our minds, that apps can be categorized in two major categories, by UI type or hosted type, and under UI type you have three kinds, and hosted types you have two kinds, and one of them has two subtypes, and what those are is very important to understand. Again, all of this will become a whole lot clearer when you actually see this in action through code.

Writing Apps

Speaking of writing apps, let's dive into it. So, by now I assume that you have setup you dev environment, etc. There are two ways to write apps. One, using NAPA. NAPA actually doesn't even require you to setup your dev environment, because NAPA is completely contained inside the browser. NAPA is an app that you can use to write apps, only for Office 365 for now, but still it's pretty cool. But, I said real developers don't use NAPA, because NAPA has a lot of limitations. I think if you are going to seriously start writing apps, you probably want Visual Studio, and you probably want a good dev environment. If you're targeting Office 365 only, and all you're doing is writing apps, then you don't need the dev environment. But I think practically speaking, I think by the end of this course you will agree with me that you want a dev environment. So, let's start by introducing ourselves to NAPA first, and then we'll dive into other topics. So.

Examining the .App File

And I'm going to make a Copy of this,and I'll rename the Copy to .zip, and I'm going to Extract the zip file. And what do I see here? That is the AppManifest.xml. There is, this looks like the same format that Word, and Excel, and PowerPoint use, the open XML formats, and inside of here, there is a wsp. Interesting. Let's Copy this wsp. Let's put it over here. And I'm going to Rename this wsp to cab, because wsp files are cab files. And for us to open this, you see here, that this is just a standard wsp. In fact, it is a sandbox solution. I thought sandbox solutions are deprecated? They're not. Let me say that like once for all to make it very, very clear, sandbox solutions are not deprecated. What is deprecated, however, is client side code. That is the server side code that sandbox solutions use. User code is deprecated, but as long as your sandbox solution is 100% declarative, that is, whatever it's trying to do, it's doing it with CAML, it is perfectly okay to use sandbox solutions. Even in SharePoint 2013. Just avoid user code. Just avoid C# code in sandbox solutions. And, in this case, if you were to try and force your way to try and create server side code, it will just not run. It'll throw an error. So, you can't do that. But anyway, what you're seeing here is that there's a manifest.xml that includes a Feature.xml, and this Feature.xml uses this Element, and this Element, and this Element, and this Element, and all of these basically contain module tags to stick all of these files inside of the SharePoint Content database. So, if you open this for instance, it is putting App.css at this location. And if you go up here, inside of the Feature, you see here that I got Pages, Scripts, Content, any of these modules are a part of this Feature, and then this Feature is a part of the package, which is the wsp. One other interesting thing you might've seen here, you see this ReplaceContent is equal to TRUE, that's pretty cool. That's a new thing they've added in 2013. So a module will override Content by default, that's useful.

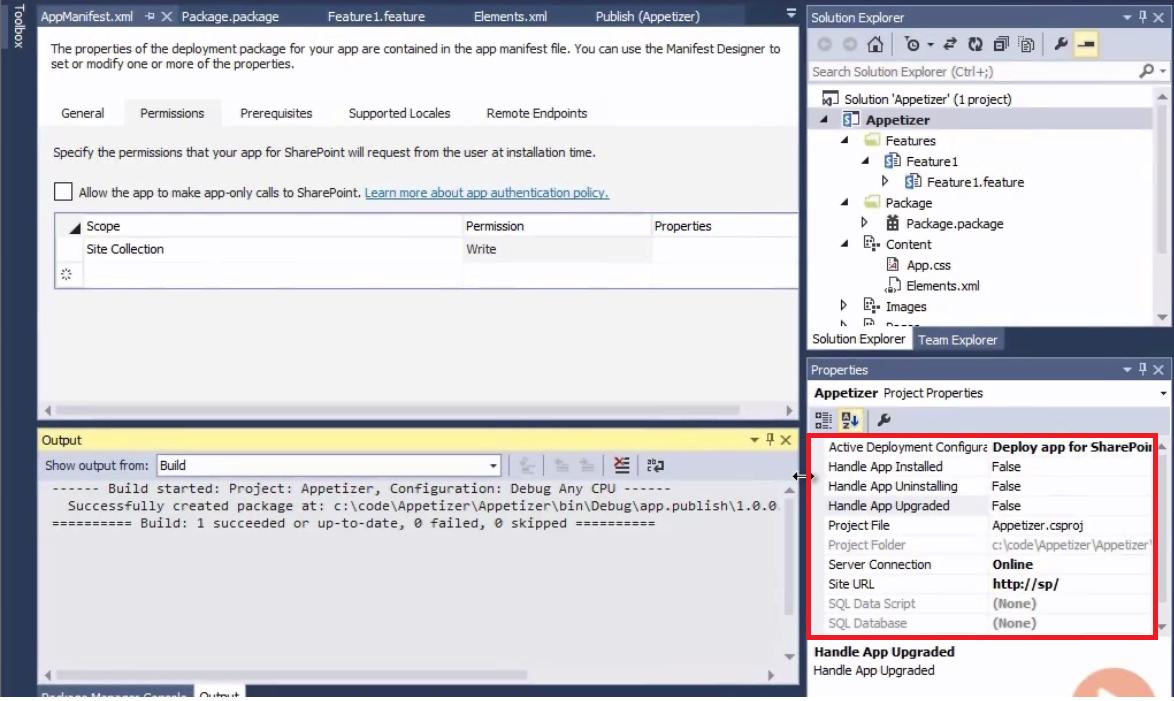
Adding Removing Apps as an End User

Okay. So it tells me that this app has been made available. Now I can go in here, and, as an end user, I can hit Refresh, and it tells me that you can't add this app here. Find out why. Because I already have it on my site. So I have to first remove it. Because I was debugging through Visual Studio, so it's already there. So let's do that. So I'm going to go ahead and REMOVE it. Click on add an app, and then I'll say choose to add the Appetizer. And it's asking for some permissions. Now, the user granting the permissions must have the permissions that the app is asking for, okay? The exception to that rule is there's something called as app only permission, which will save provider hosted apps. To grant app only permissions to an app, you have to be a site collection admin, okay? So other than that, for all other app's normal permissions, if the app asks for write permission, then the user granting the rights must have write permission. I'm going to click on Trust It, and this'll go ahead and create the child SPWeb for me, and that'll make the app available for me. It'll run just like before. Why did Microsoft go with this approach of creating a child SPWeb? Well, the basic reason over here is that the child SPWeb is a good way for me to do whatever the hell I want to do in this child SPWeb. I can create lists, I can create content types, document libraries, whatever the hell I want to do. And when I uninstall this app by simply going here and I say REMOVE, it'll clean up after itself. That's why. Basically they've deleted that app web, so everything that I created inside of that app web is now gone. Now, that's not saying that the app can never write to the host web. Imagine if you are given write access to your app. Like if you go into the permissions note up here, let's open AppManifest.xml, and we go into Permissions and I said for the scope of, let's find it, Site Collection, the permission I'm going to give is Write, okay? So if I give this permission, Write, then potentially my app could write to the host web also, and that's okay. If that's what your app needs to do, that's fine. But remember, whatever you write to the host web now, you are responsible for that cleanup yourself, okay? So it's not going to clean that out for you, so that's a very, very important consideration.

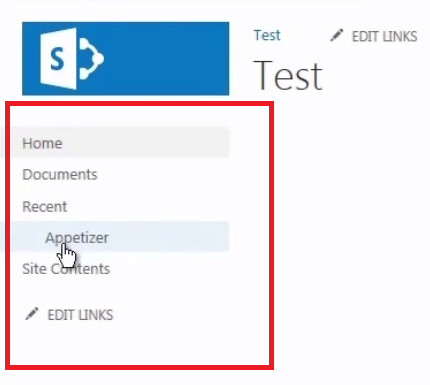
Why SharePoint Hosted Apps Don't Work with CBA or FBA

Now let's quickly go and compare the app URLs up here as well. You see here that this app URL basically looks like this. Okay. I will write ws-, but there's a star in it is my point. This is an important limitation of SharePoint-hosted apps that you cannot predict the URL of the SharePoint-hosted app until after it has been installed. And the problem this causes is that when you're using claims-based auth. When SharePoint 2013 uses claim-based auth, so it's not when, you are using claims-based auth. When, and if you use claims-based auth, that requires passive federation, and therefore http redirects, etc. SharePoint-hosted apps cannot work in that scenario. Why? Because external STS, Security Token Services, cannot redirect back to a wildcard. They need to know what URL to redirect to. So SharePoint-hosted apps do not work with things like form-based auth. They don't work with ADFS. They don't work with Azure ACS. In fact, the only thing they work with as of now, is Windows Authentication wrapped with claims, or not wrapped claims. Actually, it can't work with classic auth, so basically just Windows Auth wrapped with claims, and that's all it can work with. It cannot work with even ADFS.

Limitations of SharePoint Hosted Apps

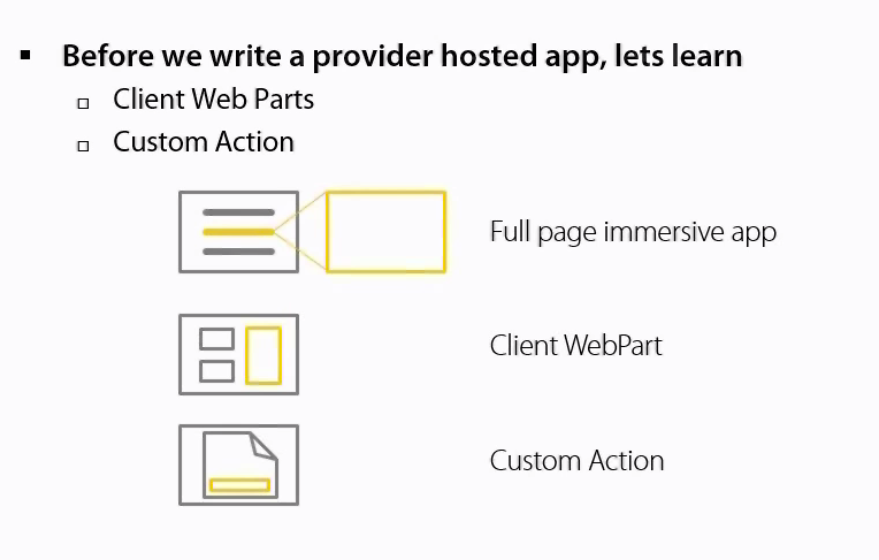


So that brings me to an interesting point about the limitations of SharePoint-hosted apps. Number one, the upgrade story, in general, is quite bad. It's not impossible to do upgrades, but the way you do upgrades in SharePoint-hosted apps, is that you handle these events. You go down here, sorry, up here, and let me expand this so we can read it, App Installed, App Uninstalling, and App Upgraded. And these are external web service calls. Now, if you are going to the extent of creating an external web service, that means you are creating, you're putting in the effort of setting up the web service as well. So if you need that complexity, why not just go with the provider-hosted apps? And I think you'll see that a provider-hosted app is actually a whole lot easier to do upgrades on, because you could have written the provider-hosted app in just ASP.NET, or MVC, and you don't need to write complex SharePoint decode to be able to handle upgrades. Secondly, whenever the app is installed and uninstalled, the AppWeb gets completely replaced. So any customizations, or any data that you put in the AppWeb, it is your responsibility to remember that, and re-create that after the upgrade. So the upgrade story is quite bad. Where it gets really bad is if you are doing upgrades that are storing things into the HostWeb, and then it's completely up to you. So when you uninstall the app, it is completely up to the developer to remove all those customizations out of the HostWeb. I would strongly recommend to you that you should not go crazy modifying the HostWeb from the AppWeb. Do what is necessary, but don't go nuts with it. The other thing I mentioned that cannot work with FBA, or ADFS, or basically anything that requires passive federation and redirect, so that's a huge limitation of SharePoint-hosted apps as of now. Maybe Microsoft will fix this in the future, but, as now, this is a limitation. I will talk about client web parts next, but one other limitation to point about client web parts in general. This is not just about SharePoint-hosted apps. Even if you are using client web parts with provider-hosted apps, client web parts have got some interesting navigation challenges that I'll explain more about when I go to client web parts. And then the full screen apps can also have some UI challenges because, let's come back here. Where's the app? Let me just quickly add the app again. Appetizer. Trust. Right. So, let's quickly go ahead and load this. Now you see here that I got the SharePoint look and feel. But the page itself that I'm using actually is, it doesn't have all the paradigms that I'm used to in a typical SharePoint site. So, for instance, if I had a custom navigation that had I built in my SharePoint site, that won't come in. The only navigation I get here is go back to the parent HostWeb, like where I launched from. Or, I have some limited ability of adding like a Help icon, and some basic navigation in here, but the apps sort of has its own navigation, and it doesn't feel like it's a part of the same SharePoint site you came from. Secondly, whenever we write code, we want to be able to give the user a URL, and the user hits the URL, and starts using the application we're writing. What happens in this scenario is that you give the user a URL to your SharePoint site, and then tell him to look for the app. Like this. And then they click on that, and then they go to the app. So there's an extra click involved.



With provider-hosted apps, you can avoid it a little bit. But, with SharePoint-hosted apps, you have to go through that extra step, because you don't know the URL ahead of time. That's the biggest issue. The third thing is that this whole area is a white canvas. It's completely in your control. That's both good and bad. It's bad because over time, which you'll see, is that a lot of people are going to invent, or reinvent the wheel in different shapes. So somebody will create a square wheel. Should I have a quick lunch here, or not. Should I create something up here, or not. People are going to create different kinds of user experiences for different apps, and there will be a lack of consistency in that. So, that's somewhat of a concern. And then, obviously one of the biggest limitations of SharePoint-hosted apps, is that they can't use any server side code. You cannot instantiate an action from the server without the initiation of the user, like a timer job, or something, that is entirely in the control of the server, and it causes something to happen in SharePoint. You can't do that with a SharePoint-hosted app. Every action must be started by the end user. What if you need server side code? If you need server side code, we do something called as we write as a provider-hosted app. Now before I dive into provider-hosted apps, I would like to talk about some more concepts about apps that are applicable to both SharePoint-hosted apps and provider-hosted apps. And those concepts are client web parts, and custom actions. So the next video I'll be extending this Appetizer example, and I'll be writing client web parts, and I'll be writing custom actions. My name is Sahil Malik. Thank you for watching, and I'll see you in the next video.

Different UI Options for SharePoint Apps



What exactly are these? When the app occupies the entire browser screen, that's called as a full page immersive app. That's the user experience we saw so far. But users are quite happy to use a web part. That's what they like about SharePoint, that they can put the page in a RIP mode, and they can drop a web part, and get some compelling functionality out of it. Apps can also work like web parts, but remember this is a pretend I'm a web part, as in it's not a real web part. It is a glorified IFrame, and that glorified IFrame in the SharePoint user interface looks and feels like a retarded web part, a web part that doesn't have all the facilities of what web parts have. Like, it doesn't have a very nice editor to it. There's no concept of two client web parts talking with each other, and so on and so forth. But you can put the page in a RIP mode, and you can drop the web part that contains an app, and we call it an app part, or client web part, it's the same thing. And the third scenario is a custom action, and as you know what a custom action is, it allows you to create links at various places in SharePoint, and you can launch an app using a custom action. So let's see this in action, how to write a client web part, how to write a custom action in our apps as a client web part, and a custom action. So again, why client web parts? Because users still like the idea of web parts. They want to be able to put a page in Edit mode, and drop a web part, or two over there. And users may want to customize the properties of what they deployed, and they want some functionality to change that way. You can do that with apps also. It's a little bit more limited than what you can do with full-fledged web parts, but you can do a little bit of customization at runtime by the user with apps as well. So, again, let's look at a demo, and understand how this works.

Using IsDlg to Reuse AppPages in Client WebParts



So for the Appetizer, I'm going to go ahead and add that is DLG equal to 1 over here. So I'm going to say IsDlg equal to 1. And if you know from SharePoint 2010, what happens is that any page that you put IsDlg equal to 1 in front of, what SharePoint will do automatically is that it will strip out all the chrome. So like the ribbon, and all of that. Basically, anything that is decorated with a class S4-NotDlg gets removed in the rending UI. So, I'm going to use a little trick, so hopefully that will make my page look a little bit nicer. So let's wait for this to deploy. And I want to see this as a client web part. I'll Open a New Tab here. And I'll go ahead and drop that web part as. Okay. So you see here that it's better than before. It's not perfect, but is better than before. I mean, I can't really scroll this very nicely. So there's a little trick how to scroll this. Basically you hold your mouse down here and scroll downwards. Not really nice, but I can at least see the values here. So, there is some scope that I have over here to be able to write one page, and use it both as the landing page full screen UI for my app, and also as a client web part. I would use concepts such as responsive design. So basically whatever resolution is thrown at me, my page will adjust itself to accommodate for that resolution. But sometimes you do need to resize the page. Sometimes you do need to be able to say, my web part is 400 pixels, and I want to get space equal to 400 pixels. How would I do that? So, the challenge that we're dealing over here, is that I cannot specify a flexible height and width to the client web part. This has to be in pixels, and fixed at the time that I'm coding. So, if I'm thrown an example where I want to be able to dynamically restructure the size of the client web part, how would I do that? The answer is, some very little JavaScript that you need to drop on your page. So, what we're doing in this JavaScript is when the document loads, get the target, which is the parent. And then we say target.postMessage, and then we send this carefully crafted up XML structure in which we give the SenderId. And the SenderId is one of the query strings that we had received. And using the SenderId, we can say resize, and then we give our width and height to the parent, so then this code will run in the parent. And as a result, it'll resize our window. So this way, when we throw different resolutions, and different dimensions of our web part, client web part to the SharePoint page, it is able to accommodate for our page. So let's go ahead and drop that as a page right here. And as you see here, that now the width and the height of the page has been adjusted. It's sort of hard to see it this way, but if I was to developer tools, it has automatically resized the size of my web part, as you can see. Cool. So, this is as far as client web parts go.

X-Frame-Options

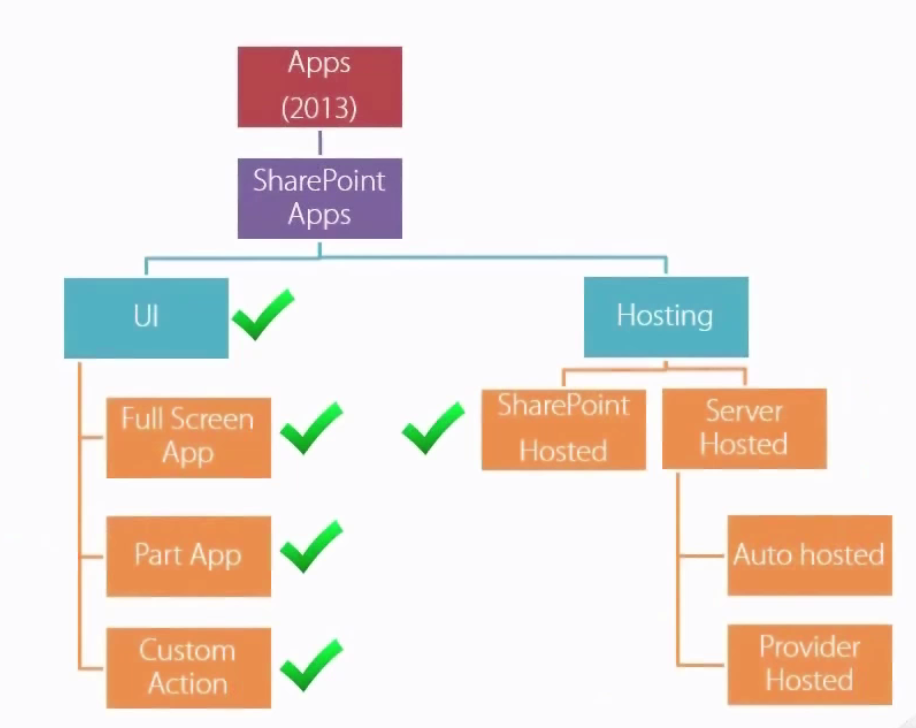
So, the important thing that we learned over here is that a SharePoint page doesn't automatically allow itself to be put inside of an IFrame. It would probably give an error. And for a SharePoint page to be able to put inside of an IFrame, you need to put this WebPartPages:AllowFraming control on any particular SharePoint page. So that's an important thing that we need to remember.

ClientWebPart Limitations and Solutions

There are some challenges as you saw with client web parts. Bookmarking becomes difficult. Being able to deep link within an app becomes difficult. Resizing the client web part becomes difficult, and there are solutions to all of these problems. So there's a well-defined HTML5-based communication API using the SenderId that you can use to get around these problems. Great.

Introduction to Custom Actions

Next, let's talk about the other kind of UI that SharePoint apps support, which is being able to launch an app using a custom action. Now, where would you find a custom action launching an app useful? Sometimes you maybe want to create a button in the ribbon that says, download this list as an Excel file. Or, convert a document to a PDF. You want to put that link in the custom action, or the ECB menu using a custom action. Those are scenarios in which you probably would find writing a custom action very useful. Custom action launching an app very, very useful. So, how exactly can we enhance our app so it has a custom action that you can use to launch an app?



Kinds of Provider Hosted Apps

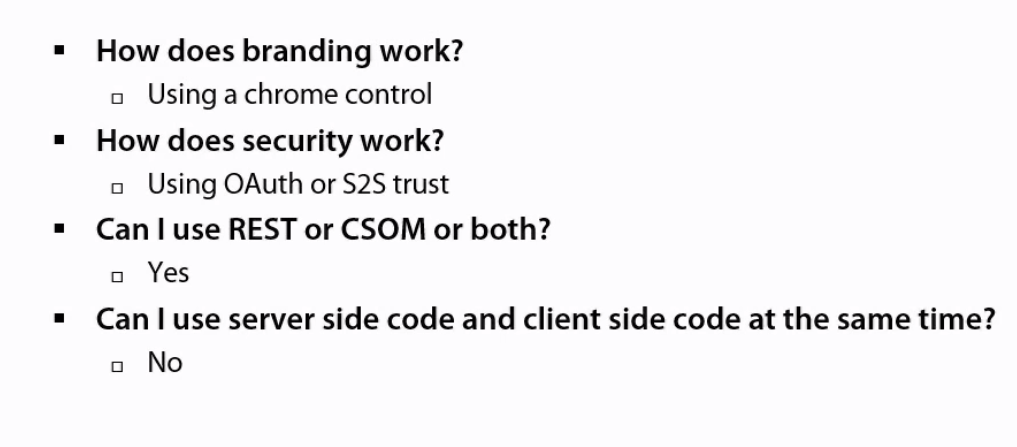
In this video, and in the next, I'll be talking about provider-hosted apps, which means apps that have server side code, or potentially a backend server that you need to set up, or is set up for you, and I'll be talking both in terms of On Premises apps, where SharePoint is installed On Premises, and in terms of Office 365, where you're running SharePoint in the Cloud. There are two kinds of server side hosted apps. One is called as provider-hosted apps where you set up the server yourself. So you would set up the server ahead of time. This is something that you would find useful in On Premises, or Office 365, where you set up the server. And the second is auto-hosted app, where a server side area is created for you every time the app is installed. So, the one major difference between provider-hosted and auto-hosted is provider-hosted gives you the facility of one installation of the server side code serving numerous installations of the app. But it does some with a little bit of an extra headache that you have to set up the server yourself ahead of time. Whereas the auto-hosted app is, if the app gets installed 50 times, then you have 50 server side areas provision. And it's really not that much of a hassle to manage those because all of those 50 areas are carbon copies of each other. So it's not like you have to go set each one of them up yourself. You basically make the process quite automated, and when the app is uninstalled, that area is removed for you. And, in this video and the next, I'll be covering again a crawl-walk-run approach. We'll be talking about the Visual Studio experience. Now behind the scenes, there's obviously a lot more going on that are things around security that are in play over here. There are things around how an app is authenticated, how is the user authenticated through the app, and all of those. I won't be covering those concepts, but let's get the basics behind us, and then I'll dive into the security aspects of the app as well.

Why Use Provider Hosted Apps?

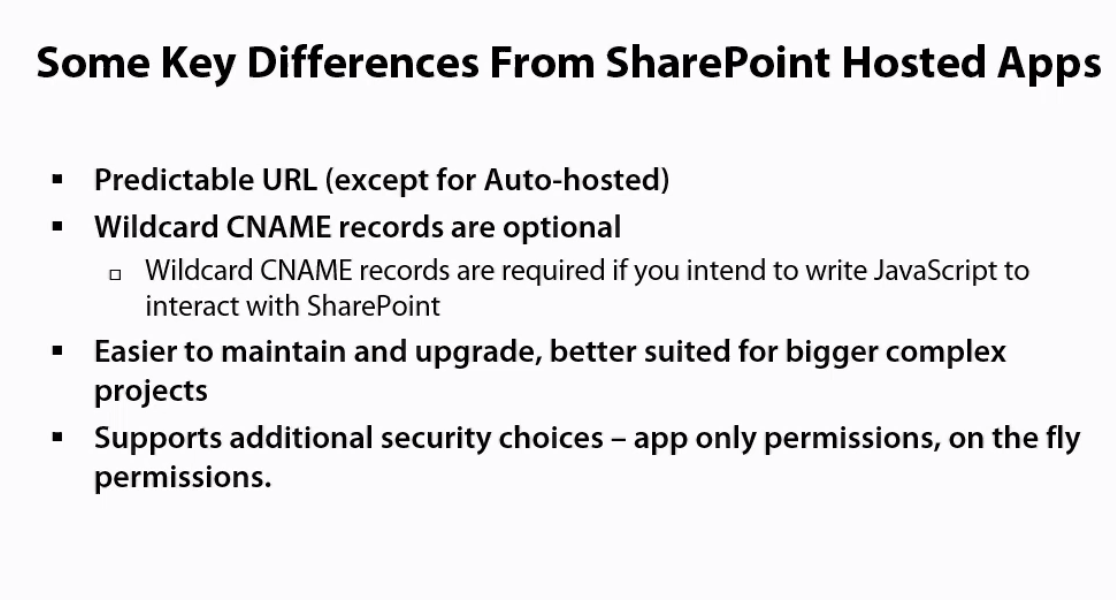
The obvious question is, why would you want to use provider-hosted apps? Well, where you need server side code, that's a pretty good reason why you would want provider-hosted apps. Example, like a timer job, or a web service, or something like that. Now, that's not the only situation where you would want provider-hosted apps. If you want to use technology, such as MVC, ASP.NET MVC, or perhaps you have existing investments, you have a big existing application like the equivalent of an Amazon or com written, and you want that to work as a SharePoint app. So you don't want to rewrite the whole thing in ASP.NET WebForms running inside of SharePoint. You want to be able to just leverage what you have, and hook into the value, and the power of SharePoint while keeping your code outside. And, I think some of us might agree that things like TDD, and better design patterns are also possible if you use other playtrons, like SharePoint can be a little bit unwielding, even when it comes to things like automated testing, and so on and so forth. So if those things are important to you, maybe you want to use apps. Maybe your existing skill set in your company is not SharePoint. Maybe it's ASP.NET, or something else, and that's where you might want to use apps, provider-hosted apps. And also provider-hosted apps open up a lot of other scenarios for you. For instance, you get a couple of security-related scenarios, like you have something called an App Only permission. So, generally speaking, when you're working inside of apps, both the user and the app must have permission to do something for an action to succeed. That is the default behavior. But, provider-hosted apps give you yet another choice where the app can do something, even if the user didn't have the rights to do it. Now this is allowed only through server side code, but that is something that, because the restriction is that only server side code can do this, it is something that is restricted to provider-hosted apps. Similarly, we've got something called as "on the fly" permissions, where the app itself didn't have certain permissions, but the app can temporarily ask the user to grant the app permissions for an hour. And as long as the user has the permissions that the app is asking for, the user can grant those permissions. These are called as "on the fly" permissions, and this is also something that requires provider-hosted apps.

How do Provider Hosted Apps Work? Client Code

The other scenario obviously is that your browser, even though it's a provider-hosted app, it wishes to use JavaScript, or some sort of client side code to be able to talk to SharePoint. In that case, the logic is actually very, very similar to what a SharePoint-hosted app would do, so you will need an AppWeb in that scenario. And your calls are proxy through the AppWeb into SharePoint. Now, there is a mechanism built over here to enable these cross-domain calls, and this mechanism is specific to SharePoint, and it is built in a way that it allows SharePoint to identify the app that was installed, and therefore not allow these requests to do more than what they should have been doing. So, the important distinction over here is that provider-hosted apps, or even auto-hosted, these apps have to choose between server side code, or client side code. So if you're using S2S trust, some people also call this high-trust apps, so if you're using high-trust apps, you're using S2S trust On Premises, then your app has to choose between server side code, or client side code. The same app cannot use both of them at the same time. Some people use a workaround where basically even if they want to use client side code, the proxy that requests from the server side. So I've written a CodePlex project called spwtf.codeplex.com, and you can download an example of that code from over there if you're interested.

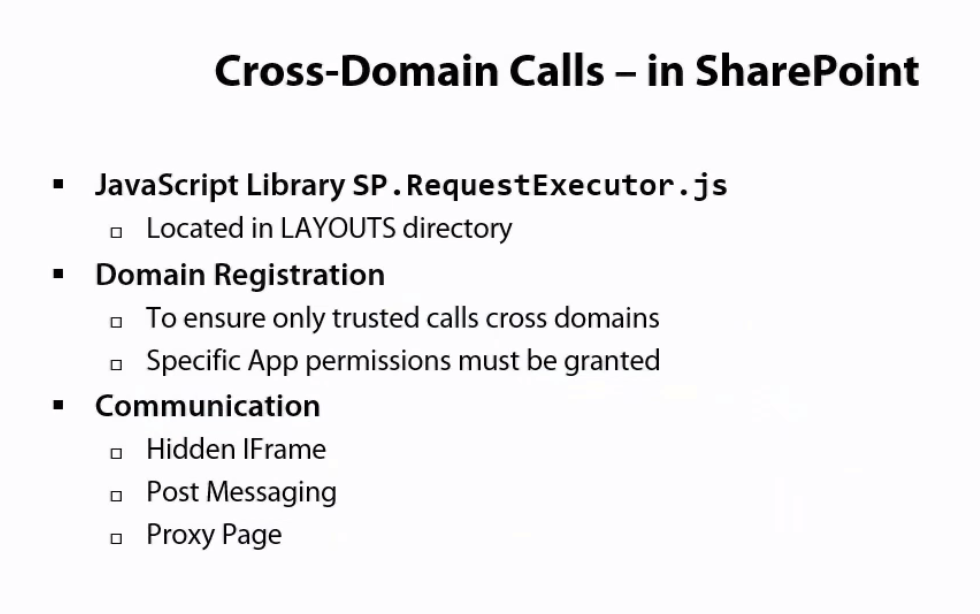


Key differences from SharePoint Hosted Apps

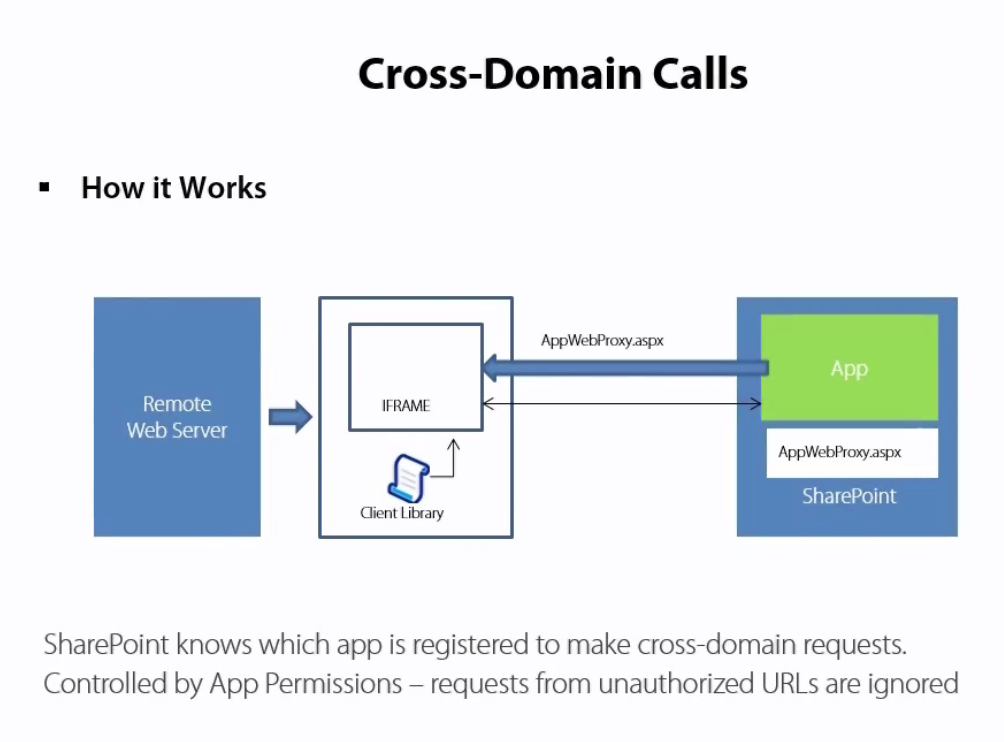


So some key differences from SharePoint-hosted apps. For provider-hosted apps, you set up the server ahead of time. The URL is predictable. So you know where the app is going to run ahead of time, and that is somewhat of an advantage, because then you don't need to set up wildcard CNAME records. Now for auto-hosted apps, that's not true, because auto-hosted apps, an app area is created for you whenever you install the app. So, in auto-hosted apps, you can't predict the URL ahead of time. But for provider-hosted apps where you set up the server ahead of time, the URL is predictable. I think provider-hosted apps, or even auto-hosted apps, are a little bit easier to maintain and upgrade than SharePoint-hosted apps. So whenever I'm writing a relatively more complex project, I prefer to use provider-hosted apps, simply because of the flexibility they offer me. Like in ASP.NET, you can do ASP.NET MVC. With TFS you can do cruise control, etc., and continuous integration. It is really nice to be able to bring all of that power into SharePoint, and therefore be able to support bigger, more complex projects that way. And there are some other security choices that SharePoint-hosted apps make available, sorry, that provider-hosted apps make available for you, like app only permissions, and on the fly permissions. So let's see some demos. What I'm going to do in this video is that I'll be showing you two demos. And through these demos, I will demonstrate three things to you. Number one, we'll be using REST API from client side code that is JavaScript only. Number two, we'll be using CSOM, Client Side Object Model, again JavaScript only. And in one case I will do it without any branding, and in the second case I will show the example with branding. And by branding, I mean the chrome control. Now, the reason I put branding in the CSOM project is because once you have the concepts clear, then we can sort of kill two birds with one stone. But remember that branding can obviously also be applied to REST API, and you can mix and match REST API in your JavaScript code, no problem. But I have created a couple of examples, and I'll be walking you through some practical scenarios that you'll see that you'll run into, and we'll build the demo as go. Some of the things won't work, and I'll explain why they're not working, and I'll explain the reasoning for why I'm doing certain things.

Demo using Cross Domain Calls in SharePoint



The way SharePoint's cross domain calls work is completely different. What we do is that we, in the app, we know the HostWeb URL. So, from the layout's directory of the HostWeb URL, we can pull in a file called as SP.RequestExecutor.js. And then with the help of this SP.RequestExecutor.js, our page basically creates an IFrame that is hidden. And this IFrame loads a page called AppWebProxy.aspx. And imagine this that your app has already been installed under certain security permissions. So let's say your app has got ability to read, but not to write. So what happens is, through SP.RequestExecutor.js, that request is funneled into the IFrame. From the IFrame, that request executes on the server. Now, at that point, the server knows that this request is coming from a certain app, and therefore it allows a certain request to succeed, or not succeed based on the permissions of the app. So, the request goes from your app, in the browser, into the IFrame, from the IFrame to AppWebProxy, and then it executes on server side, results come back, they come out of the IFrame, and there is also a return to your app. And all of this magic is done by SP.RequestExecutor.js. It sounds complex, but it really isn't.



Because SP.RequestExecutor.js hides all this complexity for us. So with that background, okay, so just keep this in mind. What's happening here, there's an IFrame, there's a client side library, and we are basically sending requests into the IFrame, through which we send them to AppWebProxy.aspx, and they run on server side. And then they come back, and that's how we get our results. So, with that in mind, let's start by looking at using REST, again JavaScript only, with the no branding, a simple example using REST API with JavaScript only. So, what I intend to do is that I'm going to write a simple REST-based code over here, targeting this site here. This is a plain vanilla team site. I've removed all the default web parts it comes with, and I'm going the write a provider-hosted app. So, I'm going to start Visual Studio. I'm going to create a New Project. And I'll create an app for SharePoint 2013, and let's call this CrossDomainREST. I'm going to hit OK. Now, I'm going to choose to make this a provider-hosted app. And typically speaking, in a provider-hosted app, when we're, especially in development environments, I'll click a Next button, and I'll choose one of these options, etc., but I don't intend to write any server side code yet. We'll do that in the next lecture. But, in this particular example, I'll just hit Finish, because I really don't care about the server side code. So let's wait for this project to get created. (Typing)