



SharePoint 2013 App Model

Succinctly

by Fabio Franzini

SharePoint 2013 App Model Succinctly

By
Fabio Franzini

Foreword by Daniel Jebaraj



Copyright © 2014 by Syncfusion Inc.
2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: Rui Machado

Copy Editor: Suzanne Kattau

Acquisitions Coordinator: Hillary Bowling, marketing coordinator, Syncfusion, Inc.

Proofreader: Morgan Cartier Weston, content producer, Syncfusion, Inc.

Table of Contents

The Story behind the <i>Succinctly</i> Series of Books	7
Information is plentiful but harder to digest	7
The <i>Succinctly</i> series	7
The best authors, the best content	8
Free forever	8
Free? What is the catch?	8
Let us know what you think	8
About the Author	9
About this Book	10
Chapter 1 Introducing the SharePoint 2013 Development Models	11
Farm Solutions	11
Sandboxed Solutions	12
App Model	12
Client API	14
Summary	15
Chapter 2 Preparing the Development Environment	16
Office Developer Tools for Visual Studio	17
Sign Up for the Office 365 Developer Site	18
Napa	18
Summary	21
Chapter 3 The New App Model Architecture Overview	22
Introduction	22

Hosting	23
Entry Point	26
App Scoping	27
Web Scope	27
Tenant Scope.....	28
Web Scope vs. Tenant Scope	28
App Web	29
Summary.....	29
Chapter 4 SharePoint-Hosted Apps	30
The Project.....	30
App Manifest	35
Summary.....	39
Chapter 5 Provider-hosted Apps	40
The Project.....	40
App Manifest	44
Summary.....	45
Chapter 6 Security Model Overview	46
Claim-Based Authentication.....	46
OAuth in SharePoint	46
App Authentication	47
App Authorization.....	48
Summary.....	51
Chapter 7 Client-Side Object Model (CSOM).....	52
Introduction	52
Managed CSOM	53
How to Use	54
Website Tasks	54

List Tasks	56
List Item Tasks.....	58
User Tasks	60
JavaScript CSOM	62
How to Use	64
Website Tasks	64
List Tasks.....	65
List Item Tasks.....	68
Working with the Web Host	70
Summary.....	71
Chapter 8 REST/OData Service	72
Run REST Calls in SharePoint Apps.....	73
Introduction to OData in SharePoint	74
Usage Examples in JavaScript	78
Reading a List	78
Creating a New List.....	78
Editing a List	78
Summary.....	79
Chapter 9 Cross-Domain and Remote Service Calls.....	80
Cross-domain calls	80
Remote Services Calls.....	81
Summary.....	84
Chapter 10 Creating User Experience (UX) for Apps	85
Look and Feel	85
Using Default Styles.....	85
Using the Chrome Control	87

App Shapes	90
Immersive Full Page	90
Client Web Part.....	90
Custom Action.....	97
Summary.....	100
Chapter 11 App Deployment.....	101
Introduction	101
Publishing	101
SharePoint-hosted	103
Provider-hosted.....	103
App Catalog	105
Office Store	109
Summary.....	110

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!



About the Author

Fabio Franzini is a Microsoft SharePoint MVP, senior consultant, software engineer, and trainer. He specializes in SharePoint, ASP.NET, scalable web solutions, and anything else that revolves around the Microsoft web platform.

Passionate about all aspects of the web, Franzini consistently utilizes the latest technologies because he believes that innovation is the real added value in everything developers create and offer.

Franzini has been a [Microsoft Ignite Group](#) instructor for SharePoint 2013 and 2010, and has delivered the first courses in Italy for a Microsoft partner since the Beta 2 version of SharePoint 2010.

He is a speaker at many conferences, maintains a blog at <http://www.fabiofranzini.com>, and can be followed on Twitter [@franzinifabio](#).

About this Book

This book's purpose is to give a quick overview of the new development model for apps for SharePoint 2013 for both its on-premises version and its cloud version (better known as Office 365—[SharePoint Online](#)).

In this book, we will start with a basic introduction to SharePoint development models where we will learn how to create apps using the new app model and Visual Studio 2013. We will then do a brief overview of the security model applied in SharePoint 2013.

Next, we will see how our app can communicate with SharePoint using the client-side object model (CSOM) or the representational state transfer (REST) API. Finally, we will learn how to create the graphical interface of our app.

The book takes a practical approach, complete with code examples that illustrate the simplicity of this new model compared to the classic development approach in earlier versions of SharePoint.

Chapter 1 Introducing the SharePoint 2013 Development Models

The use of apps is now incredibly common. People use them every day on their smartphones, tablets, desktop operating systems, and even social networks. While using the SharePoint Solution model, there is no way to define a precise border to implement an app. With a solution, we can create lists, workflows, Ribbon control extensions, event receivers, web parts, and more. Each artifact might be independent of others (depending on the logic that you create, of course). A workflow is completely independent with respect to an event receiver or a web part.

It is true that, by working together, you can implement a concept of an app, but the Solution model was created to extend SharePoint rather than create apps. The concept, more like an app in earlier versions, can be assimilated to a web part that reads or writes data in lists or libraries (this is very risky, but it's just an example).

With the introduction of SharePoint 2013, Microsoft has reinvented this concept and is now offering a new development model for a new concept of apps and extensibility. First, let's recap the current development patterns that SharePoint 2013 offers us:

- Farm solutions
- Sandbox solutions
- App model

Farm Solutions

Farm solutions are how Microsoft has enabled developers to install customizations in SharePoint since the 2007 version. Technically, these are cabinet files with a .WSP extension. Internally, a SharePoint Solution Package (WSP) contains the dynamic-link library (DLL), possibly ASP.NET WebForm pages, user controls, images, Cascading Style Sheets (CSS), JavaScript files, and various resource file element.xml that go to inform how SharePoint will have to go to install all of these things.

The farm solution pattern is very powerful because it allows us to fully extend SharePoint capabilities. We can install files inside the SharePoint installation folder, use the server-side object model that SharePoint provides, run code with privileges higher than those of the current user, and expand the SharePoint engine with event receivers, blocking just the operations that the user is doing on our own logic. In short, we have full control over our SharePoint solution.

But, as always, too much power can sometimes hurt.

When we develop a farm solution, we must always consider the fact that we have the SharePoint engine in hand.

In addition to this, we have a number of things that we must do:

1. We must use objects that SharePoint offers us in the most effective way possible, otherwise we run the risk of overloading the SharePoint farm.
2. The code that we implement must be developed using the same version of the .NET Framework with which SharePoint has been developed.
3. The development environment requires that SharePoint is present locally in conjunction with Visual Studio. This means having an available virtualization system for development or a personal computer powerful enough to run a SharePoint farm.
4. In order to install our customization with this mode of installation in a production environment, we must access the farm with administrative privileges and work with PowerShell to execute a series of commands which serve both for installation and activation.

Sandboxed Solutions

This type of solution was introduced with SharePoint 2010 and its main purpose is to be able to install customizations even to users who are not farm administrators.

In order to do this and maintain a certain level of security, this type of solution is limited compared to farm type solutions. For example, you cannot deploy any type of file on the file system. The object model is limited, preventing the code from accessing the majority of objects that are logically above the Site Collection.

Sandboxed solutions are monitored, per site collection, for resource usage based on default quotas to prevent using too many system resources.

If one or more sandboxed solutions exceeds the quota that was set for the site collection, all sandboxed solutions in that site collection will automatically be stopped.

Each night a timer job runs and resets the daily resource usage quota for all Sandboxed solutions in all site collections.

Obviously, these limits are designed to keep the SharePoint farm secure and, at the same time, give a less restrictive level to installing customizations on different sites. This model was the only way to install customizations in the SharePoint Online version since it was implemented using the 2010 version.

Currently, this model has been deprecated in favor of the new app model (at least with regard to the solutions that contain server-side code).

App Model

The app model is the new model introduced by Microsoft in SharePoint 2013 which is only available for this version and SharePoint Online (Office 365). This new pattern radically changes the way you think and develop when compared to the two models described earlier.

The main idea of this model is that the code that we implement will no longer run inside SharePoint, but instead, completely on the client-side (or server-side, but hosted outside of SharePoint), whether this means within a browser or inside a web server application such as ASP.NET, PHP, JSP, Node.js, or any other technology. This is indeed a major change, as anyone who has already developed in previous versions of SharePoint can definitely understand.

The development is done using the latest client-side technologies that the web offers including HTML 5, CSS3, and JavaScript. However, it also uses technologies such as OAuth for security and OData to REST calls or the Client-Side Object Model (CSOM) to communicate with SharePoint.

In this case, SharePoint becomes a system that provides services to our app, and especially the ability to install and run them. SharePoint provides all functionality out of the box (OOTB) in authentication mode, and specific permissions have been granted to the app and to the user.

With the model of the app we have a unified model of development, making it possible to actually install applications on SharePoint 2013 rather than on Office 365, without having to recompile or modify the project.

In addition to this, Microsoft has made available a broader set of options for providing apps for SharePoint:

1. Public Office Store: Publish your app to the Microsoft Public Office to make the App Store publically available.
2. Internal Organization App Catalog: Publish your app to an internal organization's app catalog hosted on your SharePoint deployment. This makes it available only to users with access to that SharePoint site.
3. Manual installation.

Over the course of the following chapters I will dive into the course. However, let's first analyze the main topics for the new app model:

- It is easy to migrate to future versions of SharePoint; this is because the code is no longer performed in the SharePoint process.
- Developers, thanks to client-side API, may have less knowledge of the internals of SharePoint than farm solutions development.
- The ability to use not just .NET but other technologies is available, in addition to the ability to scale those applications on different systems.

An unquestionable advantage is that in order to develop an app, you no longer need to have a local SharePoint instance. Instead, you can work completely remotely, even using a Developer Subscription of Office 365.

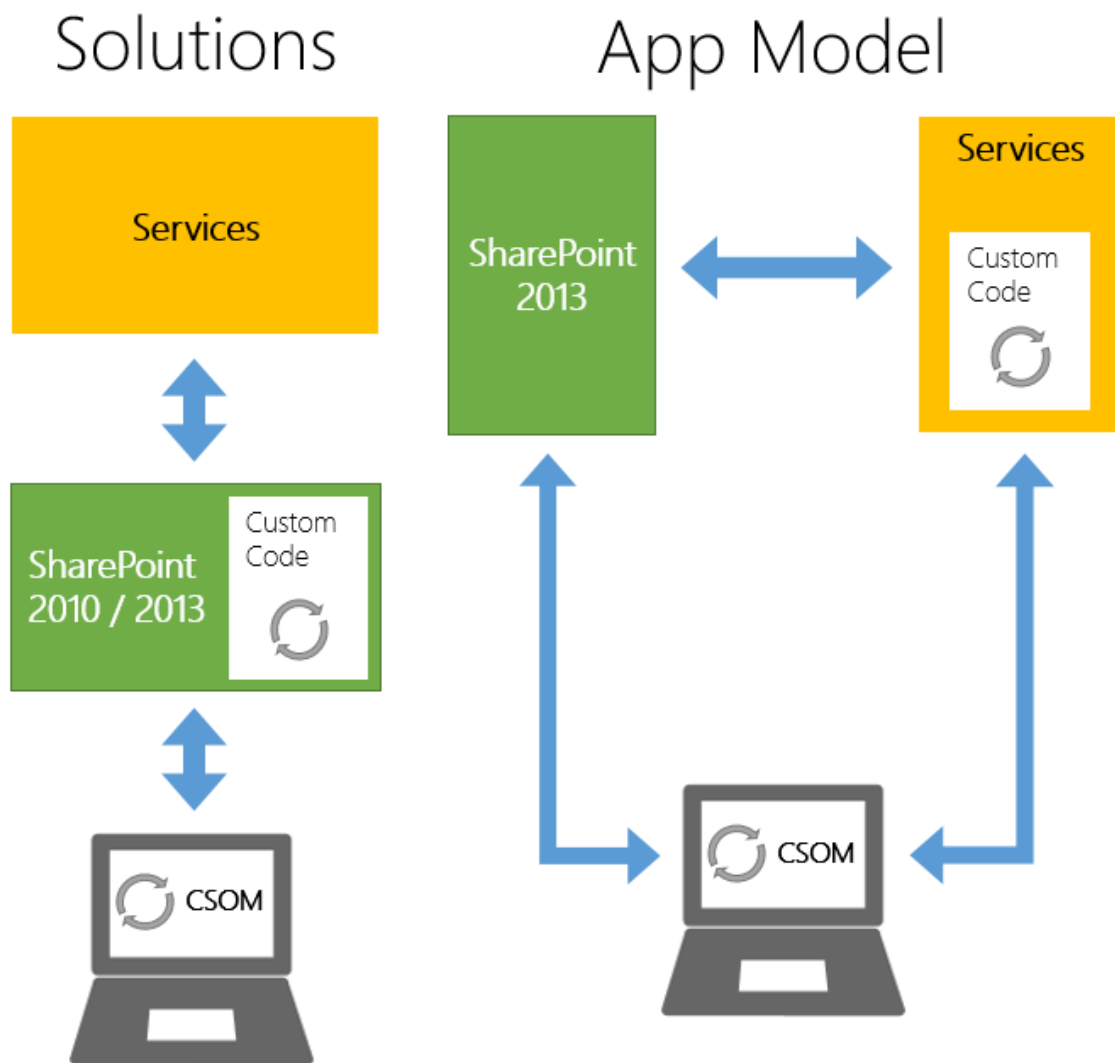


Figure 1: Solutions vs. app model

Client API

When you are unable to execute server-side code in SharePoint, the only way to communicate is to use client-side application programming interfaces (APIs) that allow us to access most of the features that SharePoint provides.

Depending on the type of application and platform that has been chosen to build the app, you can choose between:

- .NET Client Object Model (also known as a Client Object Model for Managed Code)
- JavaScript Object Model
- REST/OData

For example, if we choose to create a SharePoint-hosted app, we can decide to use the JavaScript CSOM or use Rest/OData endpoints because the main platform used is JavaScript. (Nobody forbids us to use SilverLight in a SP-hosted app, but now the plug-in for browsers has become somewhat obsolete from an architectural point of view.)

Or, if we decide to create a provider-hosted app using .NET technology, the .NET CSOM is the most appropriate. It is true that we can also use Rest/OData endpoints here, but it's definitely more convenient to use, in this context, a typed object model.

If we create a self-hosted provider using any web platform we want (PHP, Node.Js, Java, etc.), we are almost forced to use REST/OData endpoints as there is a version of the CSOM adapted to all technologies.

In the next chapters, we will further discuss these aspects.

Summary

In this chapter, we looked at models that SharePoint offers us to create customizations or apps. In the following chapters, we will address the development of apps using the new app model.

Chapter 2 Preparing the Development Environment

Another point in favor of app development for SharePoint is the creation and use of a development environment.

Since the app is available for both cloud and on-premises, the development environment can be created in two ways:

1. Creation of the classic development environment for SharePoint solutions where we need Visual Studio installed on the same machine (virtual or otherwise) where SharePoint is installed.
2. Using a developer subscription of Office 365 that allows us to have a much more streamlined system, without the burden of having SharePoint installed—which is greatly appreciated because we know that SharePoint needs a lot of resources.

Obviously, the second approach is most appropriate to what we are discussing in this book, but not for the development of farm solutions. It also allows us to begin development immediately, without losing too much time installing and configuring SharePoint properly.

In all cases where one needs to have total control of the farm being developed, it is recommended that you use a virtual machine (VM) with SharePoint installed.

You might be wondering how you can get a developer subscription of Office 365. The best way to do so is to buy it or use the one included in your MSDN subscription. You can also take advantage of a test environment for 30 days. It's your choice.

In both cases, we use a site collection that was created with the Developer Site Template that allows us to have a feature called Side-Loading enabled.



Note: Side-Loading allows us to install apps without going through an App Catalog, and then makes sure that Visual Studio can install the app directly at debugging time.

To begin creating a development environment based on Office 365, go to <http://dev.office.com/build>, scroll down, and click on “Building Apps for SharePoint.”

At this point, we must do two things: One, first make sure we have installed the Office Developer Tools in Visual Studio, otherwise we have to stop and install them. Second, we must sign up for a free trial of the Office 365 Developer Site.

Office Developer Tools for Visual Studio

The Office Developer Tools are available for both Visual Studio 2012 and Visual Studio 2013. My advice is to use the latest available version of Visual Studio so that you can use the latest features of this integrated development environment (IDE). This tool allows you to have all the project templates and project items needed to develop apps for SharePoint.

Apps for Office

Ready to build apps?

Get Visual Studio 2013 tools

[Install tools and templates](#)

Office Developer Tools for Visual Studio 2013 provides new project templates and features, so you can easily create, package, and publish your apps to the Office Store, Exchange, or a SharePoint app catalog.

If you don't have Visual Studio 2013, you can [download it here](#).

Need Office 365?

[Sign up for a free 30-day trial](#)

An **Office 365 Developer Site** makes your setup easier and helps you create, test, and deploy your apps faster.

If you have an Office 365 enterprise (E3 or E4) subscription, you can create a [Developer Site](#) at no additional charge.

Figure 2: dev.office.com site

Sign Up for the Office 365 Developer Site

If you do not have a SharePoint 2013 environment configured or a developer subscription of Office 365, you can use a trial version for 30 days to develop and test your app.

▲ Sign up for an Office 365 Developer Site

You may already have access to an Office 365 Developer Site. Here are three ways to get one:

- Are you an MSDN subscriber? Visual Studio Ultimate and Visual Studio Premium with MSDN subscriptions include access to an Office 365 Developer Site.
- Do you have a midsize business and enterprise (Plan E1 or E3) Office 365 subscription? You can purchase an Office 365 Developer Site with your subscription.
- You can either start with a **free 30-day trial**, or buy an **Office 365 developer subscription** (with or without a trial).



[or buy now](#)

Figure 3: Office 365 developer site sign up

This step requires you to create an account and buy a developer subscription to Office 365, which includes SharePoint Online, Exchange Online, and Lync Online. Once you are subscribed, follow the setup instructions.

Napa

Napa is another tool you can use to develop apps for SharePoint. This tool is distributed as an app for SharePoint (available in the SharePoint Online store for free). It allows you to develop apps that have only client-side code (HTML and JavaScript) and, therefore, are useful in contexts where you want to develop SharePoint-hosted apps and do not want to or cannot use Visual Studio.

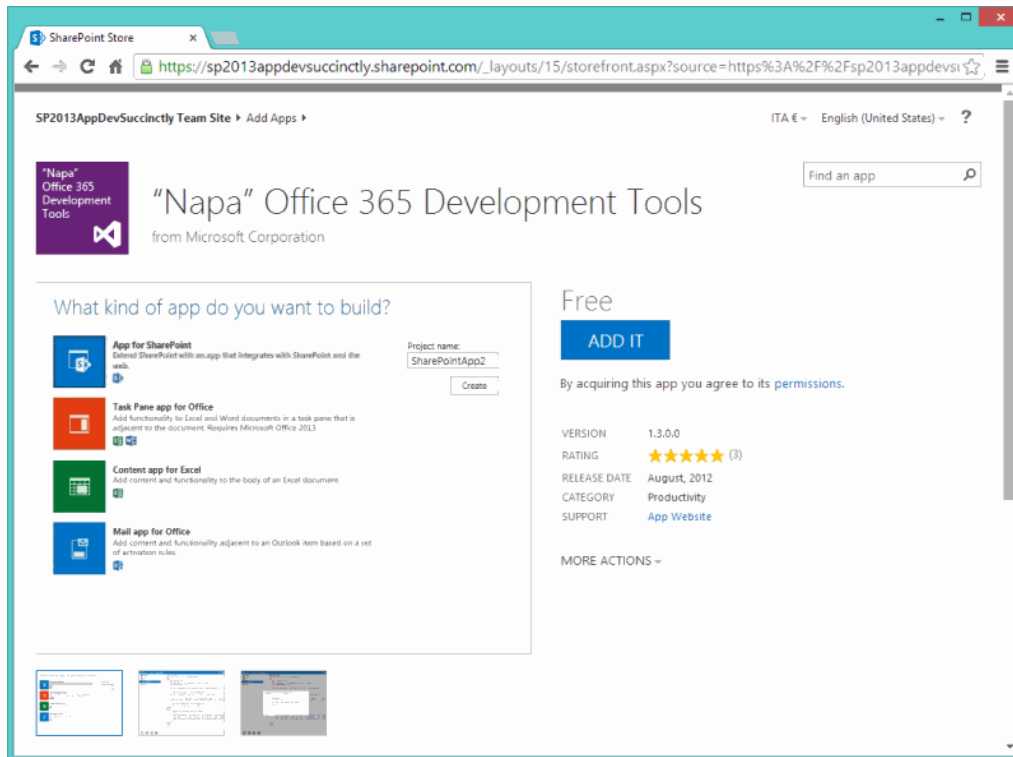


Figure 4: Add Napa app

These tools are completely web-based and allow you to not only write code but also package the app so you can then deploy it.

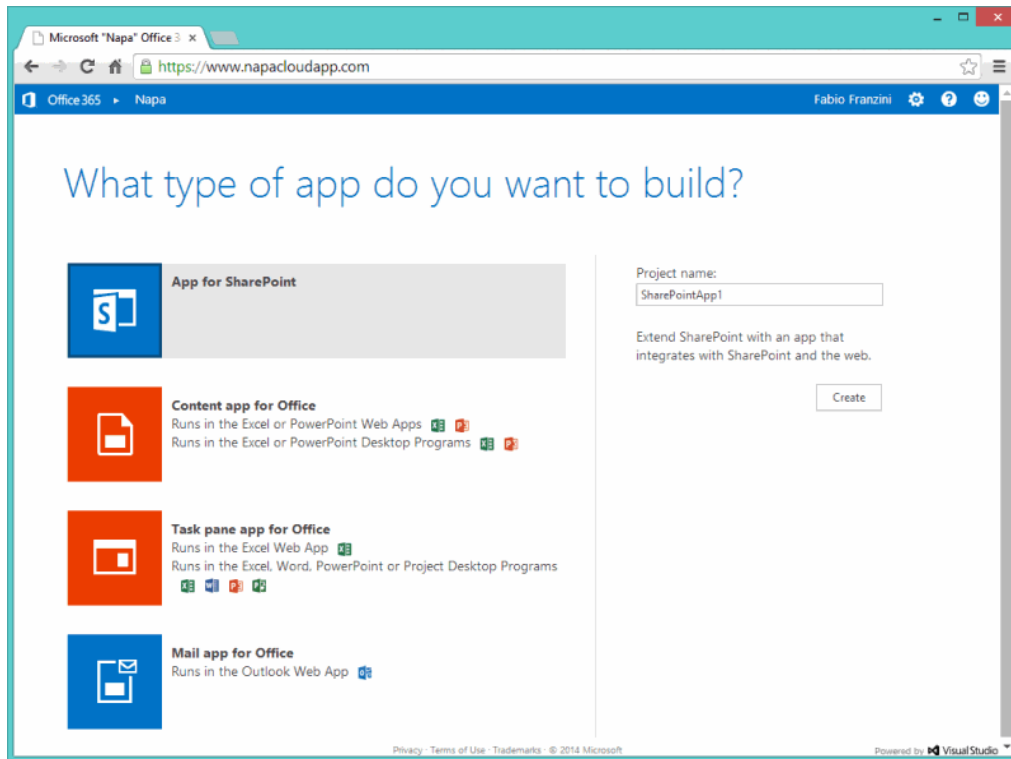


Figure 5: Create new SharePoint app using Napa

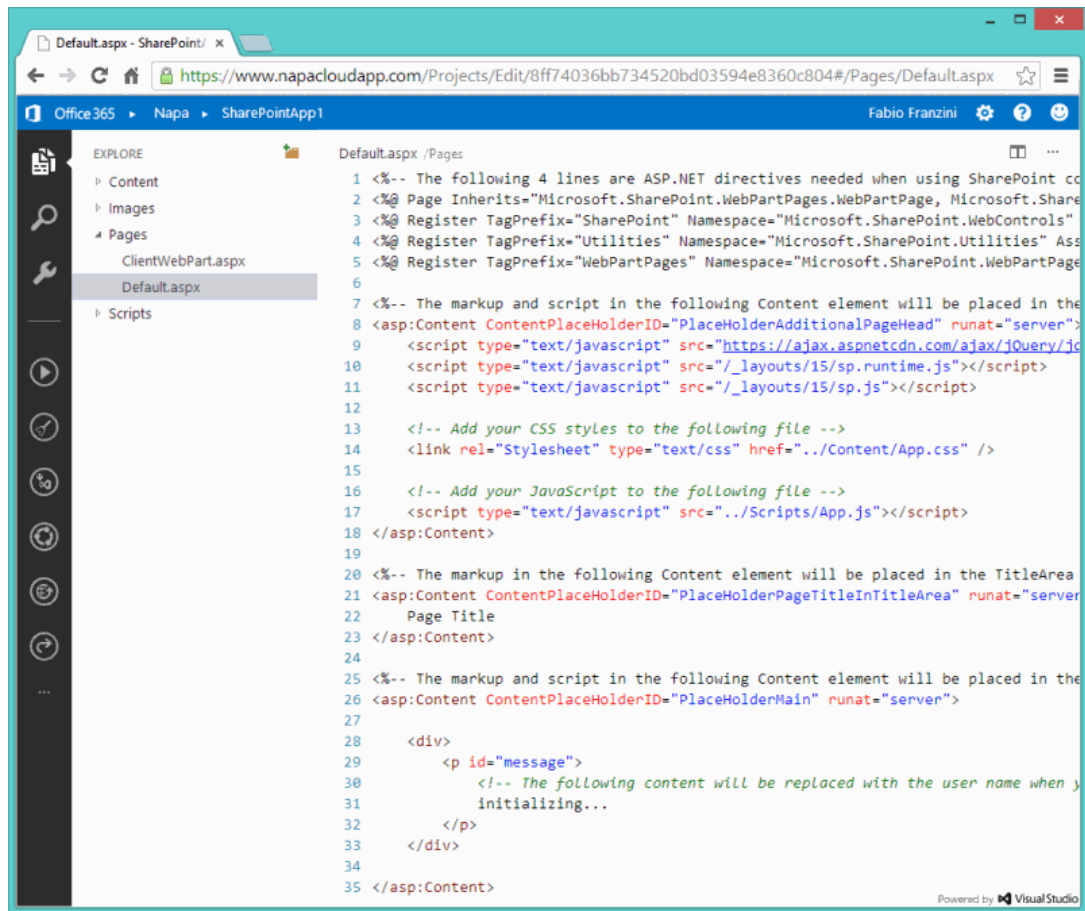


Figure 6: Modify app code using Napa code editor

Summary

In this chapter, we learned how to create, with little effort, a development environment that can create apps for SharePoint 2013. A Developer Subscription to Office 365 allows us to immediately appreciate the potential in developing an app rather than solutions, allowing the development of apps that can run both in the cloud and on-premises.

If you want to have a complete environment where you can create all kinds of apps for SharePoint, then the choice is Visual Studio with the Office Developer Tools. If you want to create only SharePoint-hosted apps and use a web-based IDE, then you can opt for Napa.

Chapter 3 The New App Model Architecture Overview

Introduction

In SharePoint, an app is technically a zip file that uses the same format as the Office documents package, except it has the file type extension “.App”.

This package contains a very important file named AppManifest.xml. This file describes our app to SharePoint with a series of metadata including title, version number, the icon, the main page, the permissions that the app should guarantee will work, the prerequisites that must be in terms of SharePoint services, the supported languages, and any external domains to which the app must have the ability to communicate, if necessary.

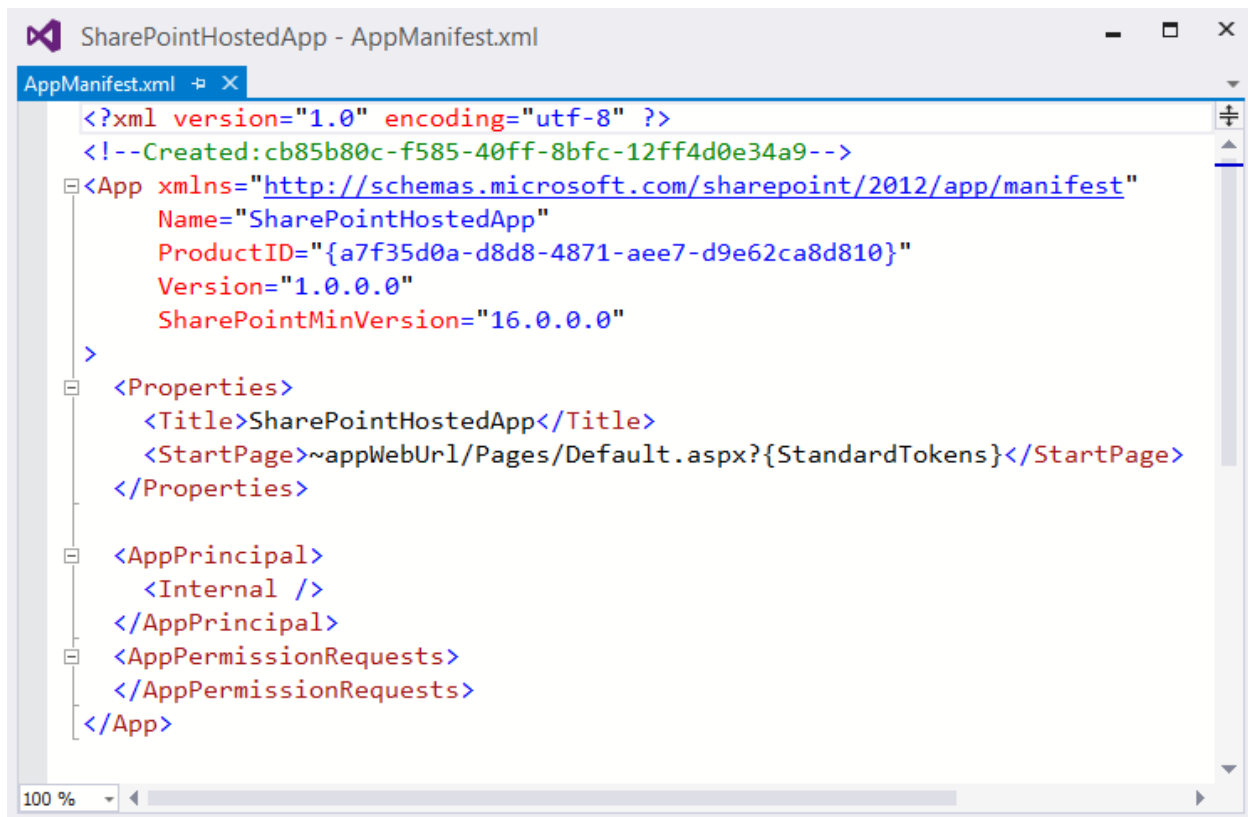


Figure 7: App manifest

In the following chapters, we'll discuss the importance of the Manifest file in more detail.

In addition to this file, the package may contain other files that need to be deployed, such as list definitions, content types or pages, and resources (as in SharePoint-hosted apps). In order to do this, the app may contain WSPs to deploy these files so that SharePoint uses the same mechanism every time.

This package installation operation, in terms of internal operations to deploy to SharePoint, is completely transparent to the user who installs the app. Figure 8 shows the App Package Structure to give you a better understanding of the concepts I have just explained:

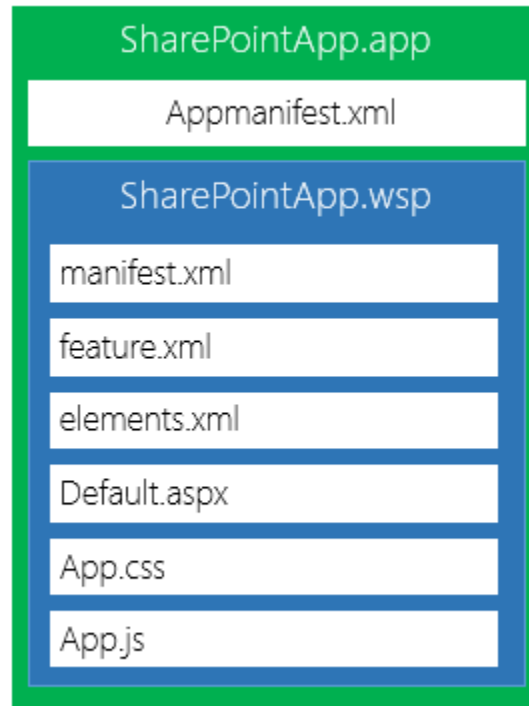


Figure 8: App package structure

To decide on the type of app architecture you need, you must understand the context in which you are developing. As in any other information system, the requirements will be a core determining factor on the architecture you use; however, the knowledge you have will be fundamental for any assessment analysis.

Hosting

When developing apps for SharePoint, we have to keep in mind the new architecture that Microsoft requires us to use.

We have two different available architectures to create a SharePoint app:

- SharePoint-hosted apps

- Cloud-hosted apps

In the SharePoint-hosted architecture, our application will reside within SharePoint and all resources (including pages) will be hosted in a special, isolated subsite that will be created for each installed instance of our app. But if we can't execute server-side code in SharePoint, how can we create applications that are hosted in SharePoint? Simple: your application logic should be created completely using client-side JavaScript. With this SharePoint-hosted architecture, SharePoint offers only the hosting service and API for us to access the functionality that they offer (but the business logic must first be performed outside of SharePoint, then client-side).

In the second case, when using the cloud-hosted architecture our applications must be hosted in a hosting service external to SharePoint; this can be an IIS site or a private or public cloud, no matter which technology you want to use to create the app. This means that we can use not only .NET but also PHP, Java, Node.js, Python or any technology stack and platform that you want.

Let us remember that we are talking about web applications that you use through a web browser. Even in this case, the application can use a special, isolated subsite hosted by SharePoint in order to read or write information through the API that SharePoint offers.

Until June 30, 2014, cloud-hosted apps were composed of two subcategories: provider-hosted apps and autohosted apps. However, the latter model is considered deprecated and has been removed as a model of development, so we will not cover it in this book.

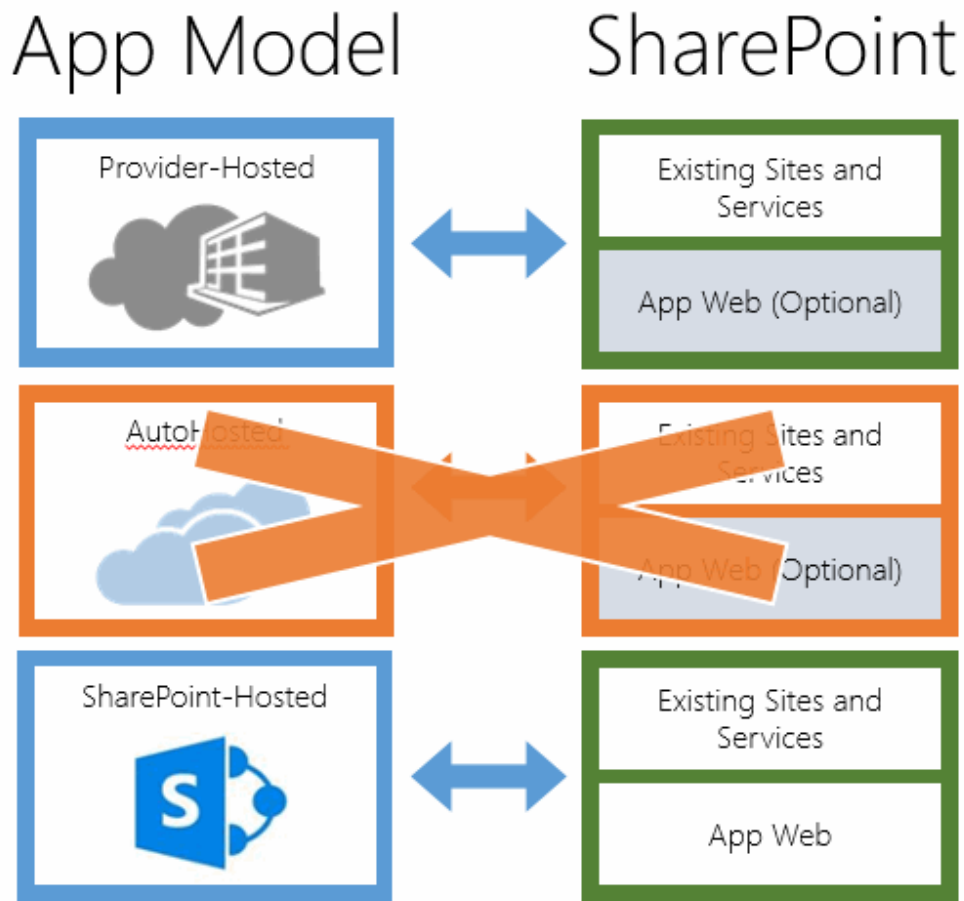


Figure 9: Different app models

At this point, which model from those available should we choose?

This is the same question we ask when we develop any app. The usual answer is that it depends on the situation. There is no one model that solves all problems. Below is a template that helps to resolve the problems we might have:

Table 1: SharePoint-hosted vs. Cloud-hosted

Requirement	SharePoint-hosted	Cloud-hosted
Use and create new SharePoint artifacts	Yes	Yes
Use client-side code	Yes	Yes

Requirement	SharePoint-hosted	Cloud-hosted
Burden of having to manage hosting space	No	Yes
Use already developed server-side components or do things you can't do server-side	No	Yes
Port apps that are already made	Depends	Yes
Run processes that can take time and high computational resources	No	Yes
Perform validation on user-submitted data, securely and with certainty	No	Yes
Automatic management of multitenancy	Yes	No

With the model of the app we can interact with the artifacts of SharePoint under Security, as we shall see in the next chapters.

The app may access only data to which we have granted permissions.

How can we create new artifacts in SharePoint? It depends. If we are deploying lists, libraries, etc., in declarative mode, these will be available in a special, isolated site called a Web App.

Otherwise, we are forced to use APIs that are provided to us, and must therefore work in imperative mode by writing code that lets us create these artifacts.

Entry Point

When an app is made, the look and feel and how the user can interact with the app itself are very important things to consider.

The SharePoint app model lets you choose different ways to present the interface:

- **Immersive Full Page:** The interface is implemented as a webpage. It is the default mode and each app requires the presence of at least one Immersive page.
- **App Part:** Let's put the GUI of our app, or part of it, in the pages of SharePoint, just as with the web parts. An app part is simply a type of web part that is represented by the ClientWebPart class. This kind of web part is essentially a wrapper for an IFrame that would host a page of the app.
- **Extension App:** Technically this is nothing more than a Custom Action that lets you share virtually all menus in SharePoint as Ribbon controls, menus on items, etc. on the web host. When users click on the link, they will be redirected to the web app.

These shapes can be used individually or combined, allowing you ability to use the application in the most integrated way possible with the SharePoint site.

App Scoping

In SharePoint, we have two ways to install an app:

1. A site administrator installs an app in the site where you want to make available; in this case, it says that the app has a "web scope."
2. A tenant administrator installs an app in an App Catalog. In this case, the app will have a "tenant scope."

In SharePoint 2013, a tenant is a grouping of site collections that, in turn, can be categorized as follows:

- **SharePoint Online:** A set of site collections that belong to a single customer account
- **SharePoint Farm:** All site collections within a Web application, or a subset of them or a collection you site collection across multiple Web applications within the same Farm.

So how do you decide which scope to give the app?

The choice of level of scope is not made during the development of the app itself but, rather, it is chosen during installation of the app in your system.

Web Scope

When installing an app with this type of scope, SharePoint creates a new subsite called an app web, but only if your app requires it. The app is able to make available the app parts and Custom Actions that you create, in the site where it is installed (web host).

Tenant Scope

In this case, only a tenant administrator can install an app with this scope and it must do so in a previously configured special site called an App Catalog. Once this is done, the administrator may decide to deploy the app on all sites that are part of the same tenants, or on one site, or on a subset of the available sites. The tenant admin can specify on which websites the app is installed by using a list of managed paths, a list of site templates, or a list of site collections. An app that has been batch-installed can only be uninstalled by a tenant administrator. When the tenant administrator uninstalls the app, it is uninstalled from every website in the tenancy. Users cannot uninstall a batch-installed app on a website-by-website basis.

An important note regarding this operation is that when updating a batch-installed app, only a tenant administrator has permission to do it, and it is batch-updated on all websites in the tenancy where it is installed.

If an app that includes an app web is batch-installed, only one app web is created and it is shared by all the host websites on which the app is installed. The app web is located in the site collection of the organization's app catalog. When new site collections are created in the tenancy, apps that were previously batch-installed are automatically installed on the new site collection.

You also need to keep in mind something regarding the app web. In this case, you will create a single "app web" App Catalog level. This means that any data saved in this special subsite will be shared between all sites where the app was installed. You can learn more about the app web in the next section.

Web Scope vs. Tenant Scope

Table 2: Scope comparison

Scenario	Web Scope	Tenant Scope
Share the same data, independently from the site you are working on	No	Yes
App must be installed on a very large number of sites	No	Yes
App must have different data for each installation	Yes	No
Make available the app parts and Custom Actions on the site on which it is installed	Yes	No

App Web

We understand what the app web is and where it is created according to the scope that the app has. Because it is a SharePoint site it will have a URL, but it won't be a normal URL.

The URL's main function is completely isolated, even at the domain level: the app.

Its composition is as follows:

`http://[TENANTS]-[APPID].[DOMAIN]/[APPNAME]`

Where:

- TENANTS: The tenant name configured in SharePoint
- APPID: The unique identifier that is generated for each instance of the app
- DOMAIN: The domain name of SharePoint
- APPNAME: The name of the App (the value that is contained in the App Manifest file)

An example would be the following:

`https://sp2013appdevsuccinctly-8235ceb59053d4.sharepoint.com/SharePointApp1`

Where:

- sp2013appdevsuccinctly is the TENANT
- 8235ceb59053d4 is the APPID
- sharepoint.com is the DOMAIN name (in this case, it comes to Office 365)
- SharePointApp1 is the APPNAME

This app was installed on the site with the address:

`https://sp2013appdevsuccinctly.sharepoint.com`

This illustrates the difference between the two URLs. Although, remember, this is a subsite app web (SPWeb) site where the app was installed.

As we said a moment ago, the main purpose is to completely isolate the app. So, for example, you can delete the classic XSS issues. But another purpose is to enforce the permissions so that the code that will connect to the site will be isolated from the point of view of authorizations.

Summary

In this chapter, we learned about the architectural layout of the new model. We now understand three fundamental factors in the design and implementation of a SharePoint app: the hosting model, the kind of graphical interface will have in our app, and its scope.

Chapter 4 SharePoint-Hosted Apps

As mentioned in previous chapters, a SharePoint-hosted app is a type of app that is hosted on SharePoint completely within a particular subsite called an app web. Unable to execute server-side code anymore, we are forced with this type of app to create business logic with a fully graphical interface using client-side technologies such as HTML 5 and JavaScript. Actually, you can create such apps to invoke external services but we will talk about that later. Thanks to Office Developer Tools, we have at our disposal the templates that we need to create the project.

The Project

Let's discuss how to create our first SharePoint-hosted app using Visual Studio 2013.

Open Visual Studio. Create a new project by selecting the template "Office/SharePoint", then select "Apps" and select "App for SharePoint 2013":

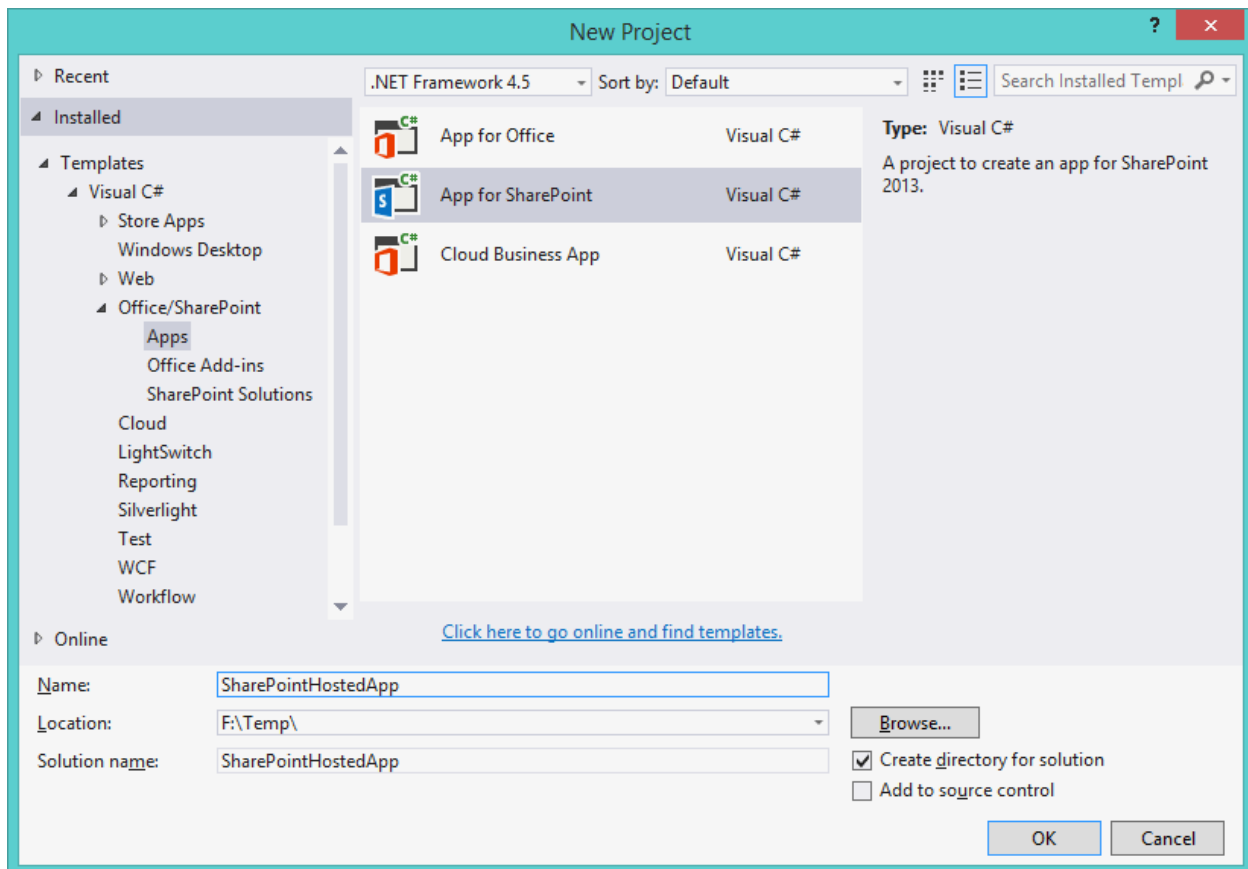
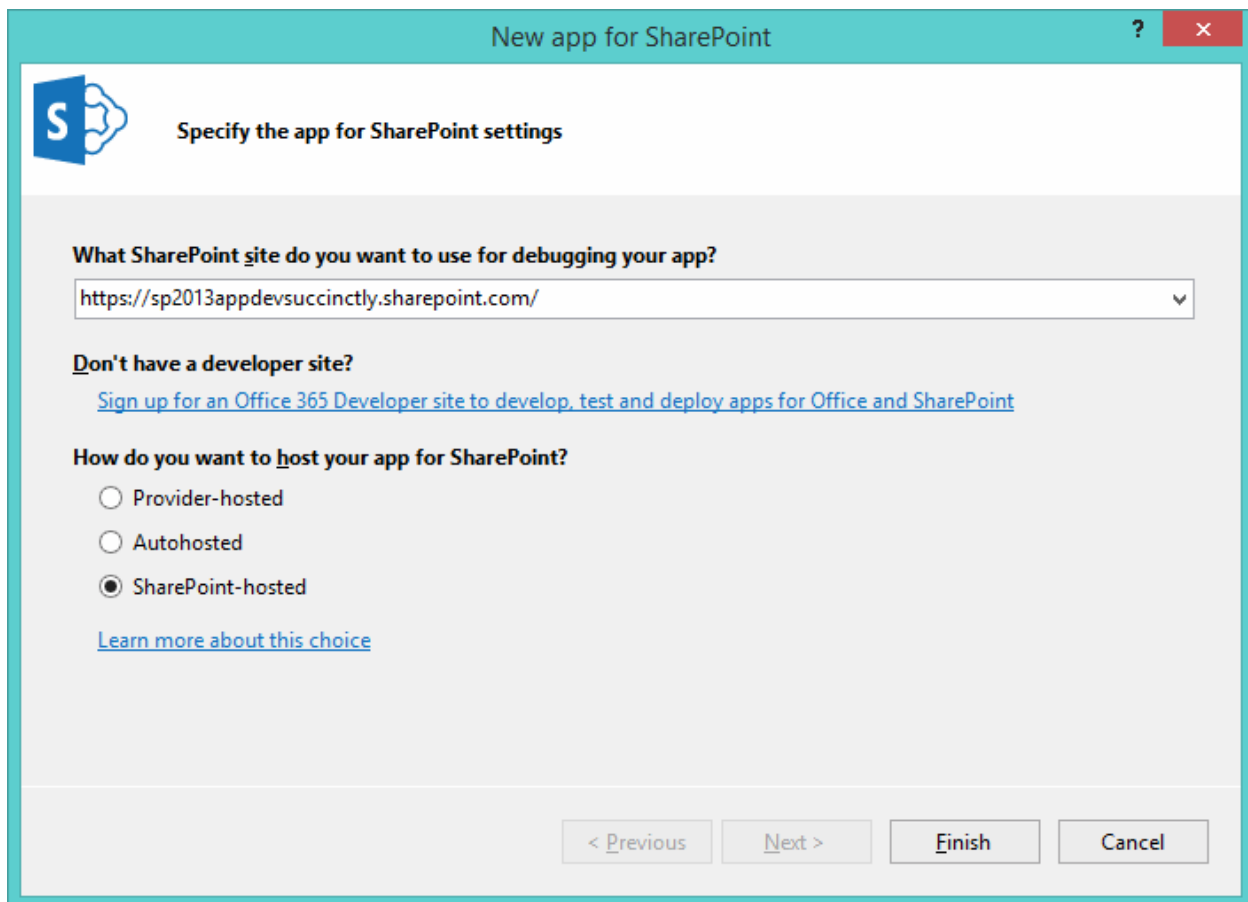


Figure 10: SharePoint app templates

As with any project in Visual Studio, give it a name and a path.

At this point, Visual Studio asks us to enter information (which can be modified later) that includes the name of the app, the site we want to perform debugging on, and the type of app we want to create. In this case, choose SharePoint-hosted as shown in Figure 11:



New app for SharePoint

Specify the app for SharePoint settings

What SharePoint site do you want to use for debugging your app?

https://sp2013appdevsuccinctly.sharepoint.com/

Don't have a developer site?

[Sign up for an Office 365 Developer site to develop, test and deploy apps for Office and SharePoint](#)

How do you want to host your app for SharePoint?

☐ Provider-hosted

☐ Autohosted

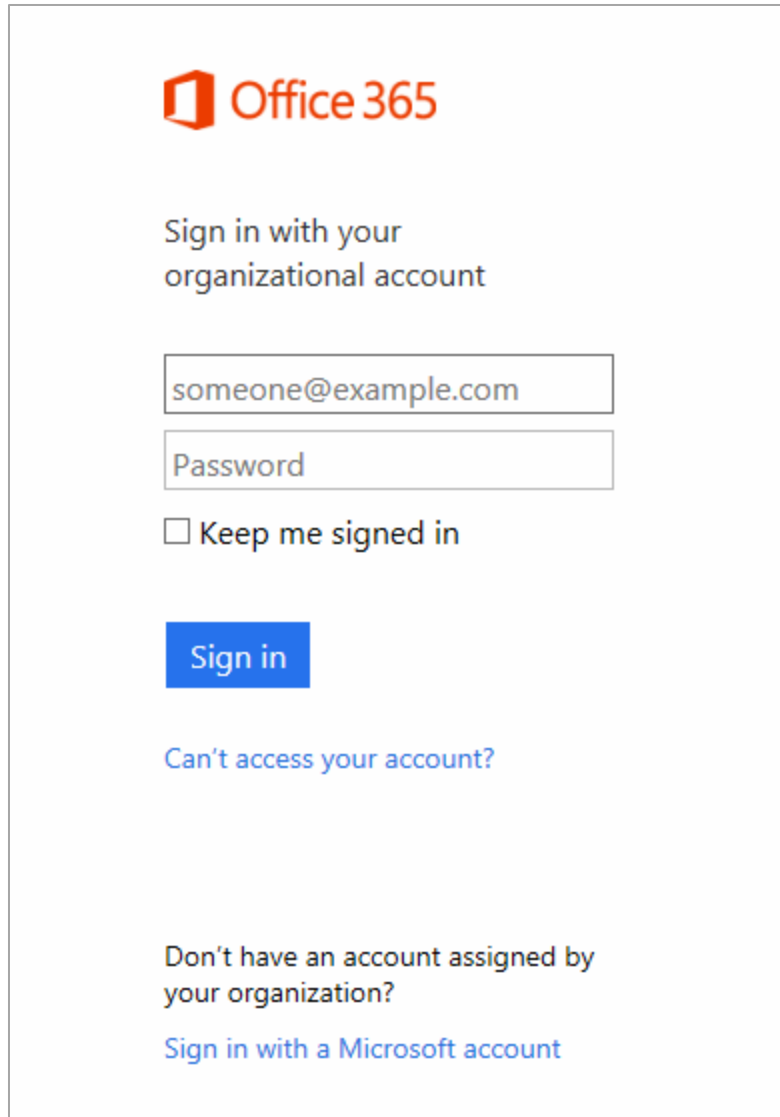
☒ SharePoint-hosted

[Learn more about this choice](#)

< Previous Next > Finish Cancel

Figure 11: Choice of hosting type

If you use Office 365, Visual Studio prompts you to log in so you can debug. You can then install the app (you can also skip this step and log in later).

The image shows the Office 365 sign-in interface. At the top is the Office 365 logo. Below it, the text "Sign in with your organizational account" is displayed. There are two input fields: the first contains the email address "someone@example.com" and the second is labeled "Password". Below the password field is a checkbox labeled "Keep me signed in". A blue "Sign in" button is positioned below the checkbox. At the bottom of the sign-in section, there is a link that says "Can't access your account?". Further down, there is a section for users who do not have an account assigned by their organization, with a link that says "Sign in with a Microsoft account".

Office 365

Sign in with your
organizational account

someone@example.com

Password

☐ Keep me signed in

Sign in

[Can't access your account?](#)

Don't have an account assigned by
your organization?

[Sign in with a Microsoft account](#)

Figure 12: Office 365 sign in

When you complete this step, we have a new project in Visual Studio for SharePoint. The structure of the newly created project is shown in Figure 13.

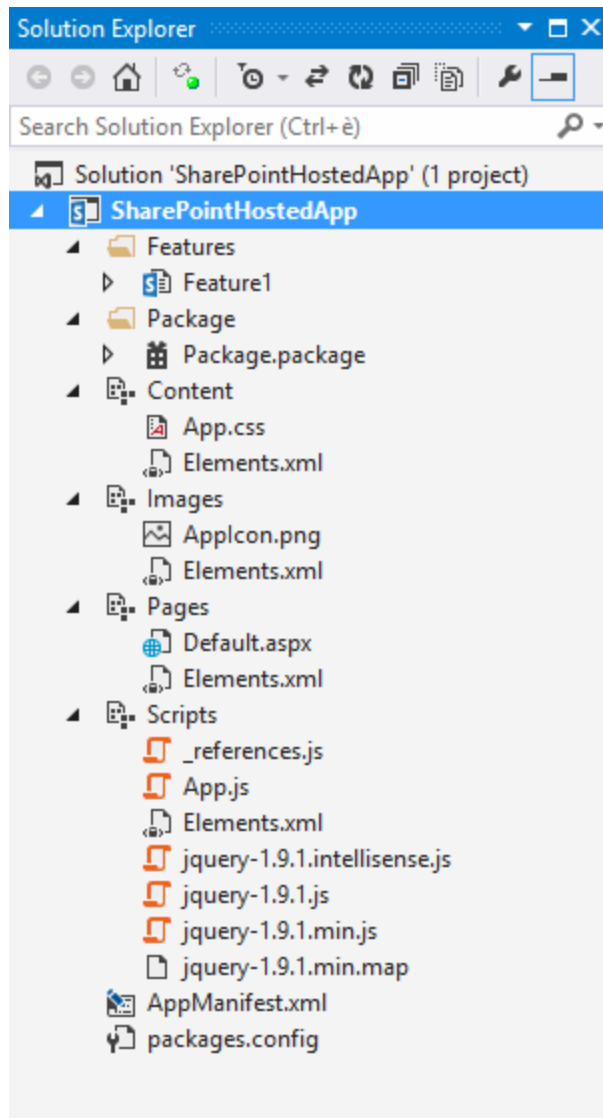


Figure 13: Solution Explorer

As you can see, there are several project items that, in turn, consist of several files. The project items called Content, Images, Pages, and Scripts are SharePoint modules.



Note: Modules are what makes it possible to deploy files within a "virtual file" (or content database) to SharePoint, so then you can make those files available through specific URLs.

These modules contain all the files that must be installed within SharePoint and, more specifically, within the web app that will be created. In the module Pages, you can find the Default.aspx page that contains markup that implements this small app example. The Scripts module contains all the JavaScript files that need the app to work. In the Content and Images modules, there are CSS files and images using this app. By opening the file App.js, we can find this code:

```
'use strict';

var context = SP.ClientContext.get_current();
var user = context.get_web().get_currentUser();

// This code runs when the DOM is ready and creates a context object which is needed
// to use the SharePoint object model.
$(document).ready(function () {
    getUsername();
});

// This function prepares, loads, and then executes a SharePoint query to get the
// current user's information.
function getUsername() {
    context.load(user);
    context.executeQueryAsync(onGetUserNameSuccess,
        onGetUserNameFail);
}

// This function is executed if the above call is successful.
// It replaces the contents of the 'message' element with the username.
function onGetUserNameSuccess() {
    $('#message').text('Hello ' + user.get_title());
}

// This function is executed if the above call fails.
function onGetUserNameFail(sender, args) {
    alert('Failed to get user name. Error:' + args.get_message());
}
```

This code uses the CSOM in JavaScript to do something very simple that helps you retrieve the username of the user who is currently running this app.

Once you have successfully recovered your username, use jQuery to set the value within a paragraph of the Default.aspx page; otherwise, you receive an Alert that contains the error message.

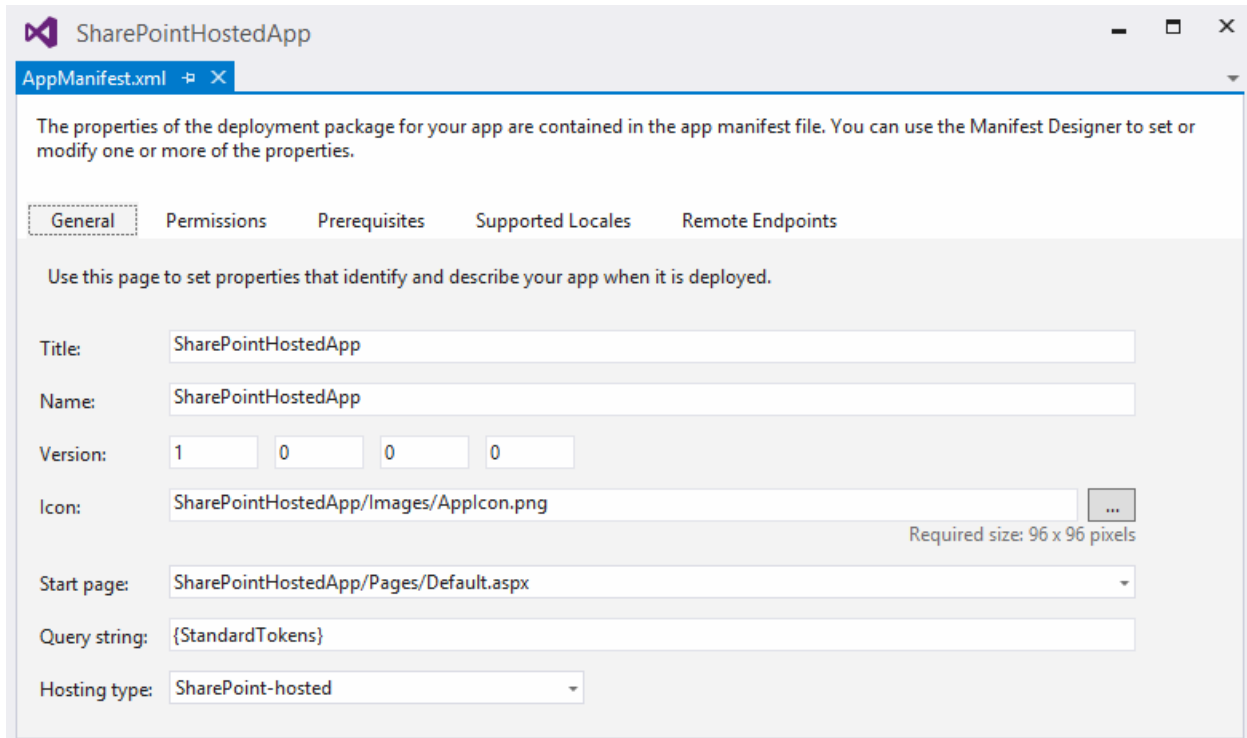
We will analyze the CSOM in the next chapters, but remember that because we are in a SharePoint-hosted app, we can only use the JavaScript implementation.

A highly interesting thing to note here is that there are two things common to SharePoint solutions, features and packages: for example, we can go to add a list or a Custom Action or other artifact.

Why? Because all of the artifacts of SharePoint, such as Lists and Content Types, can also be published through the classic system (WSPs) but this time in a completely transparent way. In the next chapters we will learn how.

App Manifest

There is another file in the project: AppManifest.xml. This file, which we will find even in provider-hosted apps (as mentioned in previous chapters) is the Application Manifest:



The screenshot shows a window titled "SharePointHostedApp" with a tab for "AppManifest.xml". The window contains a description: "The properties of the deployment package for your app are contained in the app manifest file. You can use the Manifest Designer to set or modify one or more of the properties." Below this are five tabs: "General", "Permissions", "Prerequisites", "Supported Locales", and "Remote Endpoints". The "General" tab is active and contains the following fields:

- Title: SharePointHostedApp
- Name: SharePointHostedApp
- Version: 1.0.0.0
- Icon: SharePointHostedApp/Images/AppIcon.png (with a note: Required size: 96 x 96 pixels)
- Start page: SharePointHostedApp/Pages/Default.aspx
- Query string: {StandardTokens}
- Hosting type: SharePoint-hosted

Figure 14: General tab

What we see is the graphic designer of the App Manifest. We can open the file and display its XML contents:

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Created:cb85b80c-f585-40ff-8bfc-12ff4d0e34a9-->
<App xmlns="http://schemas.microsoft.com/sharepoint/2012/App/manifest"
  Name="SharePointApp1"
  ProductID="{6d3a5861-3593-4af7-9673-df0d350d8ae9}"
  Version="1.0.0.0"
  SharePointMinVersion="16.0.0.0">
  <Properties>
    <Title>SharePointApp1</Title>
  </Properties>
  <StartPage>~appWebUrl/Pages/Default.aspx?{StandardTokens}</StartPage>
</App>
```

```

<AppPrincipal>
  <Internal />
</AppPrincipal>

</App>

```

As we can see from the graphic designer, there is a lot of information that we can add.

If we go to the Permissions tab, we will see the configuration of the app's security; this is where we can also indicate what permissions the app must have in order to work:

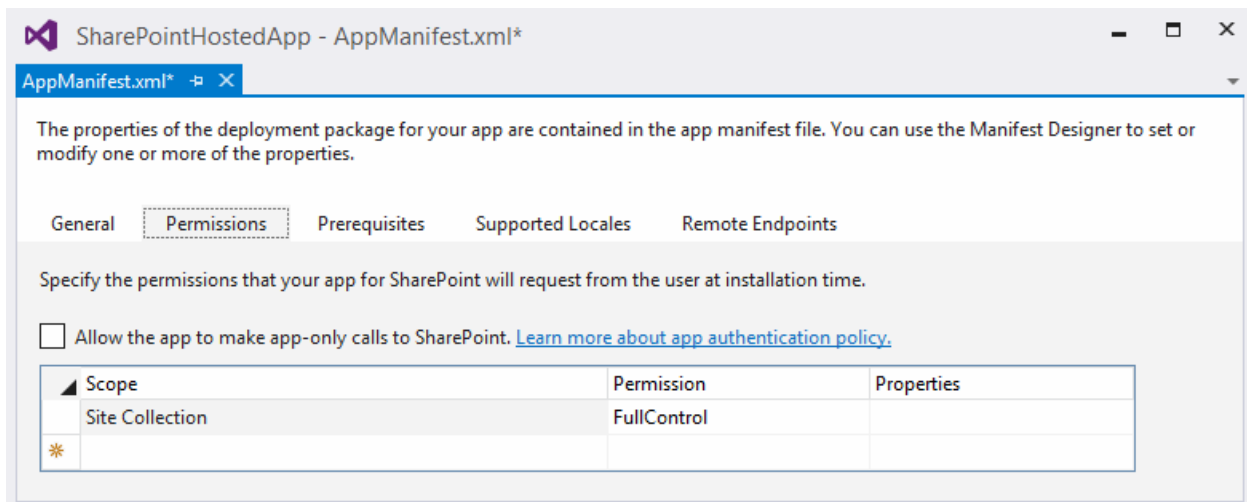


Figure 15: Permissions tab

The Prerequisites tab allows you to configure all prerequisites (SharePoint Services, for example) that should be guaranteed to work in the app:

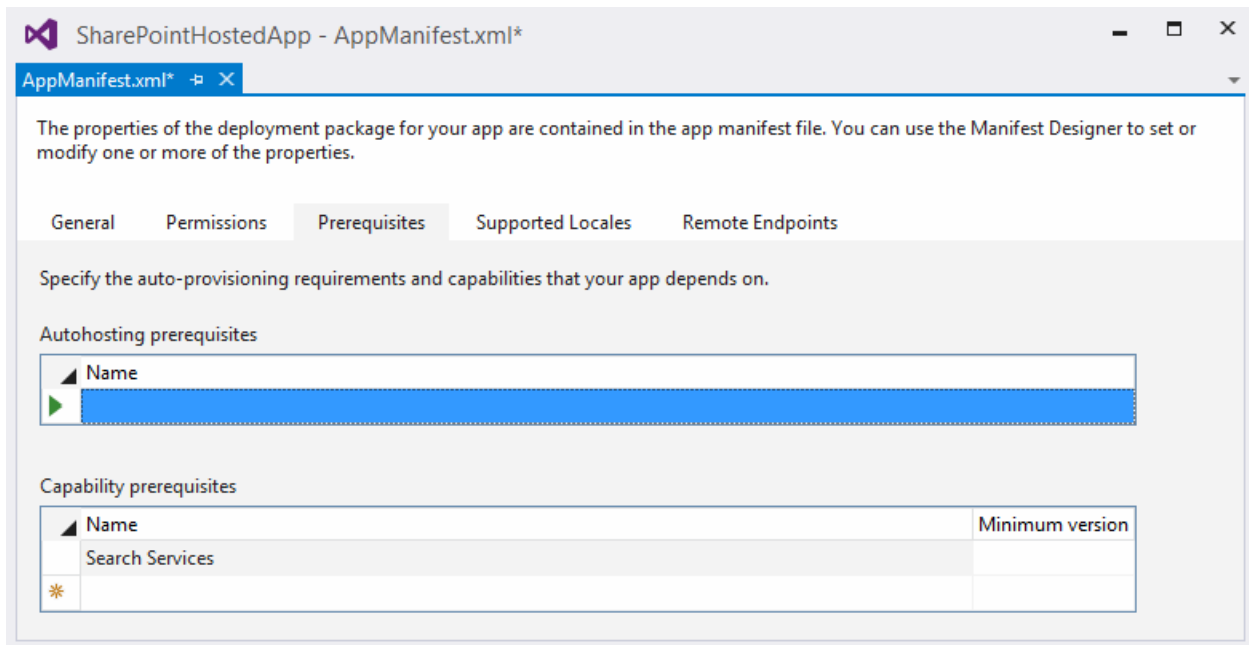


Figure 16: Prerequisites tab

The Supported Locales tab serves to indicate the languages in which the app will be available. For each language, it will generate a resource file which can then be used in your pages.

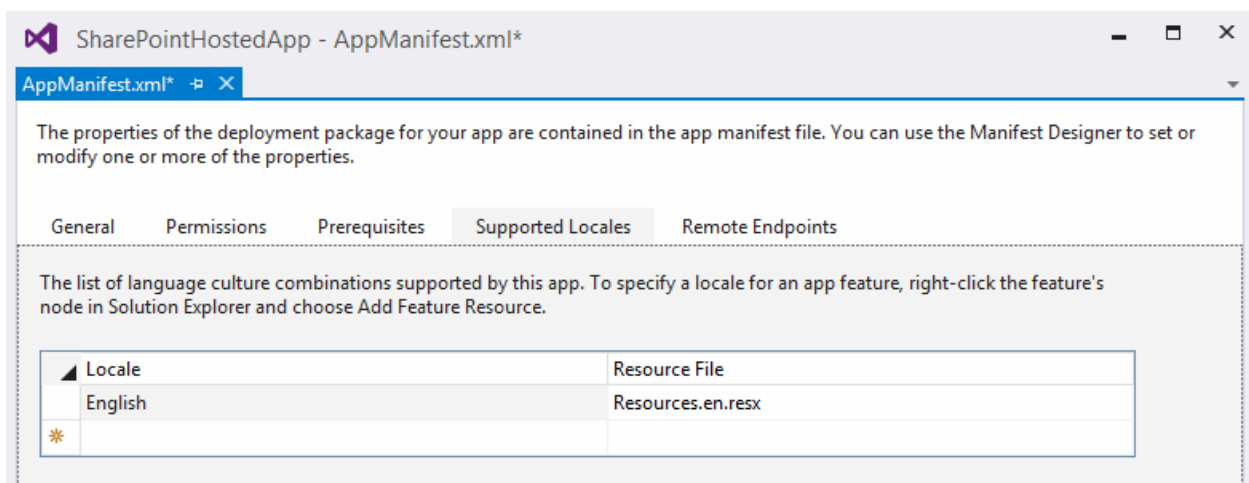


Figure 17: Supported Locales tab

The Remote Endpoints tab indicates the external services to which the app will eventually connect in order to read or consume data. SharePoint checks cross-domain calls and blocks those that are not authorized if they do not exist within the App Manifest. We will see more details on this in the next few chapters.

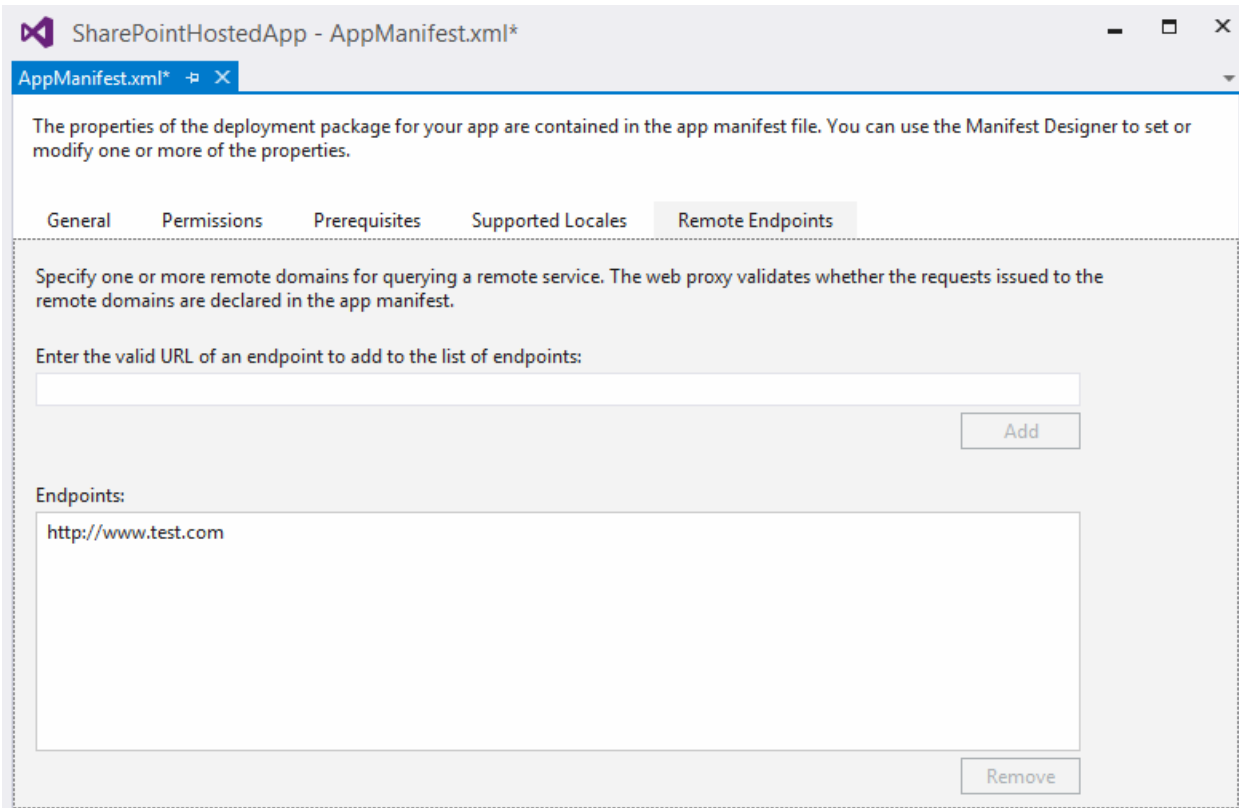


Figure 18: Remote endpoints

The result is as follows:

```
<?xml version="1.0" encoding="utf-8" ?>
<!--Created:cb85b80c-f585-40ff-8bfc-12ff4d0e34a9-->
<App xmlns="http://schemas.microsoft.com/sharepoint/2012/App/manifest"
  Name="SharePointHostedApp"
  ProductID="{6d3a5861-3593-4af7-9673-df0d350d8ae9}"
  Version="1.0.0.0"
  SharePointMinVersion="16.0.0.0">
  <Properties>
    <Title>SharePointHostedApp</Title>
    <StartPage>~appWebUrl/Pages/Default.aspx?{StandardTokens}</StartPage>
    <SupportedLocales>
      <SupportedLocale CultureName="en" />
    </SupportedLocales>
  </Properties>

  <AppPrincipal>
    <Internal />
  </AppPrincipal>
  <AppPermissionRequests>
    <AppPermissionRequest
```

```
Scope="http://sharepoint/content/sitecollection"                Right="FullControl" />
  </AppPermissionRequests>
  <AppPrerequisites>
    <AppPrerequisite Type="Capability"
ID="132084D8-5DA6-4EAB-A636-3ADF44151846" />
  </AppPrerequisites>
  <RemoteEndpoints>
    <RemoteEndpoint Url="http://www.test.com" />
  </RemoteEndpoints>
</App>
```

In the XML syntax, the address of the start page is the first part, the value **~appWebUrl**. This value will be replaced during installation with the value corresponding to the URL of the app web being created.

Summary

In this chapter, we learned how to create and structure the SharePoint-hosted app. As mentioned previously, this app must be developed using only client-side technology such as JavaScript. It is possible to insert into your project artifacts such as lists, Content Type, Modules, etc., declaratively. This will automatically be installed in the app web that will be created during installation.

Chapter 5 Provider-hosted Apps

A provider-hosted app for SharePoint is a type of app that provides the ability to execute server-side code but not within SharePoint. It is, therefore, necessary to host the web application that is part of the app using an application server such as Internet Information Services (IIS) or any other according to the technology used. This is because we are no longer forced to use .NET for the web part and, therefore, are free to use any language or platform we want.

The Project

At this point, Visual Studio asks us to enter information (which will be modified later) that includes the name of the app, the site on which we want to perform debugging, and the type of app we want to create. In this case, choose “provider-hosted” as shown in Figure 19:

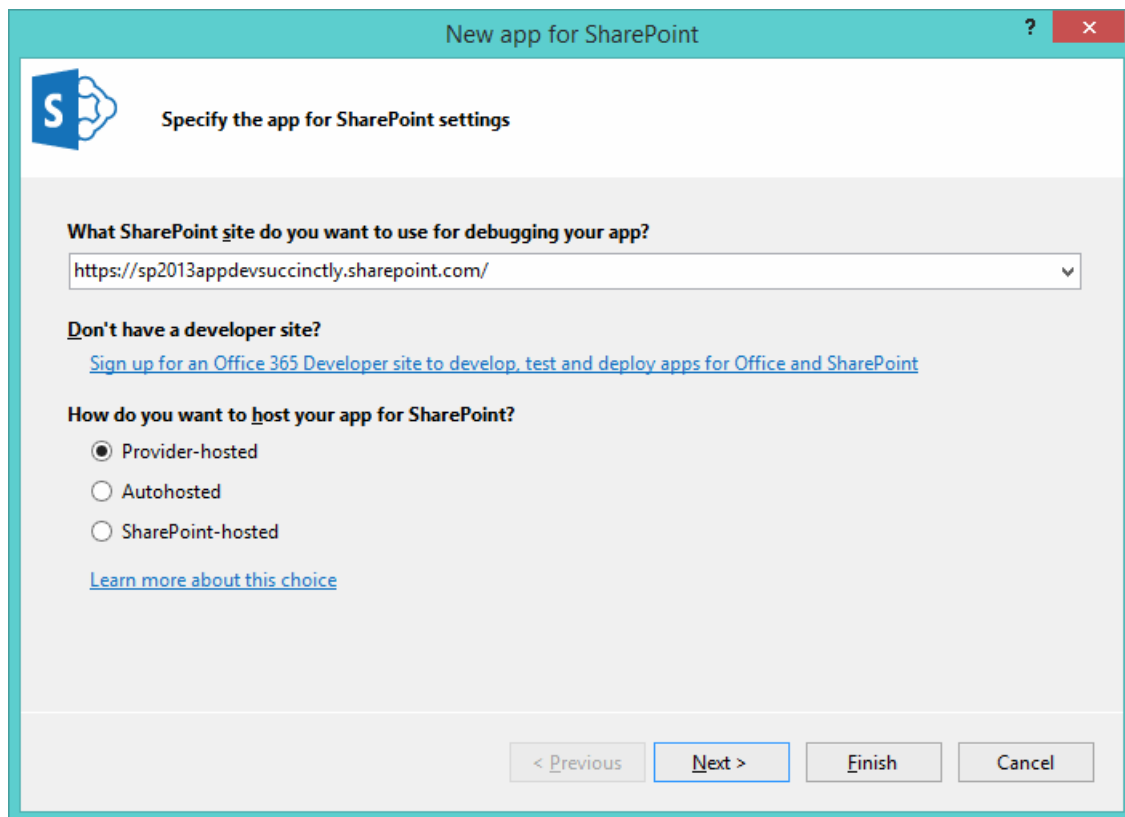


Figure 19: Choice of host type

In this screen, Visual Studio allows us to choose the type of web project we want to use for the creation of our app. We can choose ASP.NET Web Forms or ASP.NET MVC:

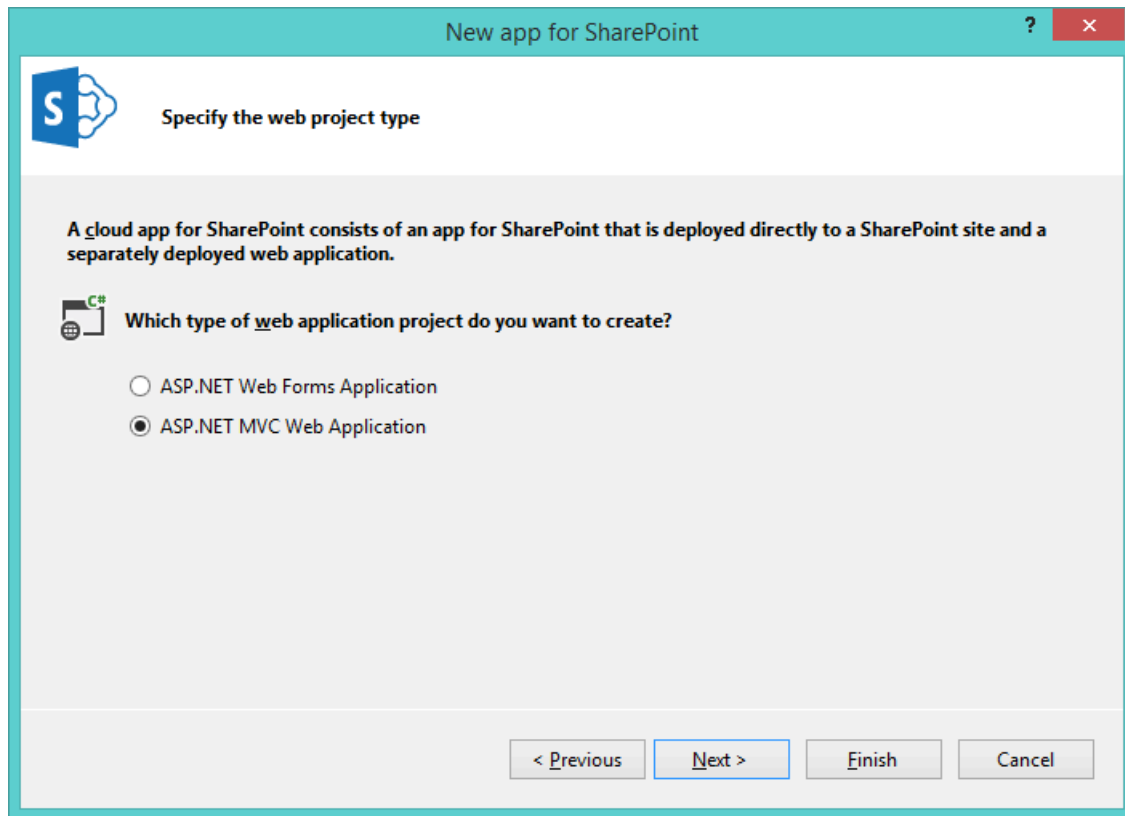


Figure 20: Choice of web application project type

Here, we need to specify the way our app will authenticate to SharePoint using SharePoint Online. Our choice is either the Access Control Service (ACS) of Azure or we can set up a digital certificate.

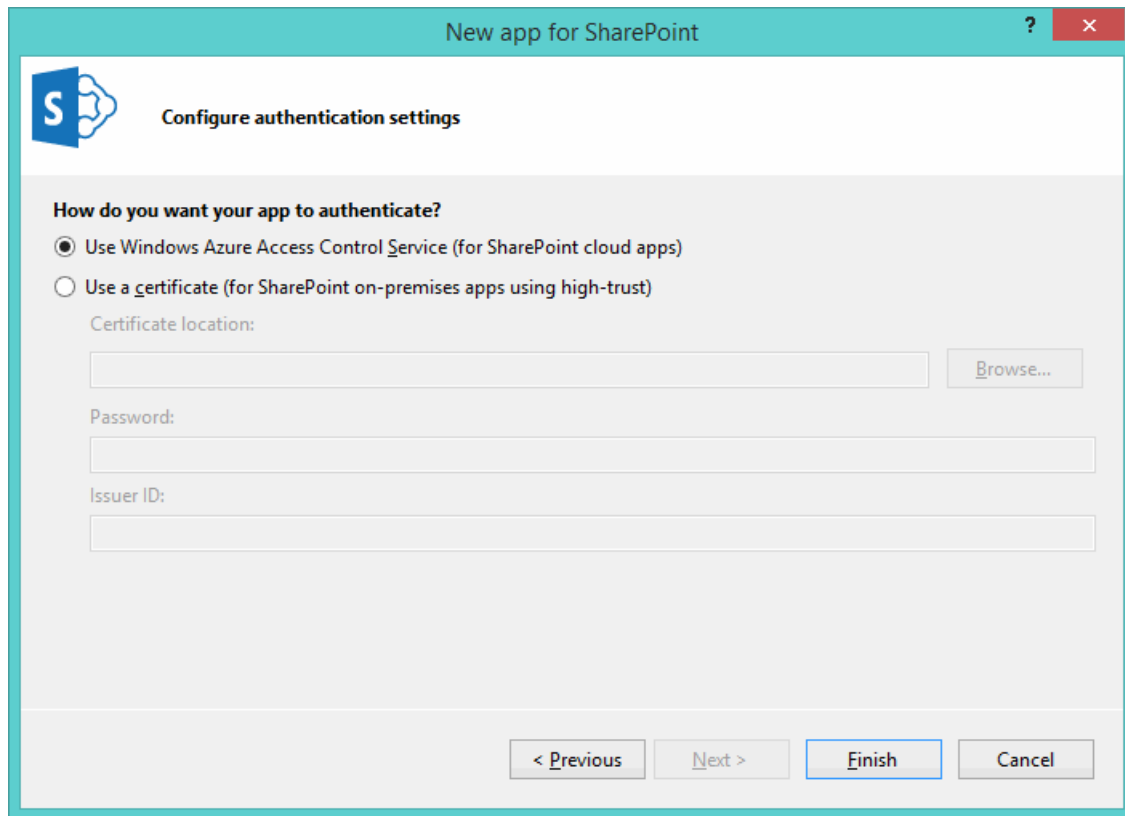


Figure 21: Choice of app authentication type

Once you click Finish, the solution is presented with two projects. The first is the real project for SharePoint apps, the second is the web project that contains the code of our app.

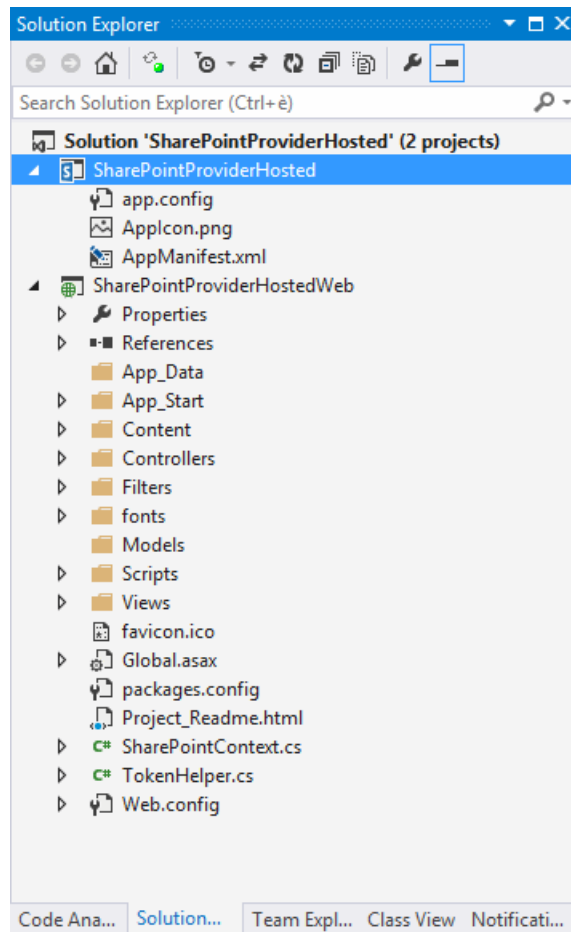


Figure 22: Solution Explorer

This web project, an ASP.NET MVC type, contains some interesting files.

The first file is the **SharePointContext.cs** file that allows us to create a new ClientContext for use in CSOM, starting from the current http context.

The second is the **TokenHelper.cs** file, which is used internally by the SharePointContext class and to initiate a mechanism to obtain an Access Token valid for requests under the OAuth authentication.

Inside the HomeController controller, we can see how it uses this mechanism in order to have a valid ClientContext and make a call to SharePoint using the CSOM Managed.

```
var spContext = SharePointContextProvider.Current
    .GetSharePointContext(HttpContext);

using (var clientContext =
    spContext.CreateUserClientContextForSPHost())
{
    if (clientContext != null)
    {
```

```

    // CSOM Code here
}
}

```

The **SharePointContext** class has several methods already made in order to create different **ClientContext** based on the scope of use:

- CreateUserClientContextForSPHost
- CreateUserClientContextForSPApp web
- CreateAppOnlyClientContextForSPHost
- CreateAppOnlyClientContextForSPApp web

These would create contexts to connect to the app web or HostWeb, either using only the credentials of the App or combined with user credentials, making it very easy to use.

App Manifest

With regard to the AppManifest, there aren't any major changes compared to what has been said for SharePoint-hosted apps. As you can see, the difference is the type of hosting, which is set to provider-hosted. However, everything else is the same as described in the [App Manifest section](#) of Chapter 4.

The screenshot shows the 'SharePointProviderHosted' application manifest editor. The 'General' tab is selected, showing fields for Title, Name, Version, Icon, Start page, Query string, and Hosting type. The Title and Name are both 'SharePointProviderHosted'. The Version is set to 1.0.0.0. The Icon is 'SharePointProviderHosted/AppIcon.png' with a note that the required size is 96 x 96 pixels. The Start page is 'SharePointProviderHostedWeb/'. The Query string is '{StandardTokens}'. The Hosting type is set to 'Provider-hosted'.

Figure 23: General tab

In XML view, we can see how the Start Page is identified:

```
<Properties>
  <Title>SharePointProviderHosted</Title>
  <StartPage>~remoteAppUrl/?{StandardTokens}</StartPage>
</Properties>
```

The value of **~remoteAppUrl** will be replaced with the domain name where the app will be hosted, this being the published app.

Summary

In this chapter, we learned how to create a provider-hosted app as well as how it is structured. We chose to create an ASP.NET MVC web project, but you can also create one using ASP.NET Web Forms.

Visual Studio provides these two templates for the web project but you can use any web development framework you want.

Chapter 6 Security Model Overview

Claim-Based Authentication

With SharePoint 2013, the default authentication mode is the claim-based authentication mode, which has been present, but optional, since SharePoint 2010. The classic authentication mode has been deprecated.

With the introduction of claim-based authentication, you can support several authentication and authorization scenarios, such as app requirements, all thanks to the presence of a Security Token Service (STS).

In claim-based authentication, by default, the authentication provider generates a security token that contains a set of claims that describe the user's authentication, and when a user logs on to a website, the user's token is validated and then used to access SharePoint. The user's token is a secure token issued by an identity provider.

All of this is also possible by using a protocol called OAuth.

OAuth in SharePoint

OAuth is an open protocol for authorization. OAuth enables secure authorization from desktop and web apps in a simple and standard way. OAuth enables users to approve an app to act on their behalf without sharing their user name and password.

For example, it enables users to share their private resources or data (contact list, documents, photos, videos, and so on) that are stored on one site with another site, without users having to provide their credentials (typically, user name and password).

OAuth enables users to authorize the service provider (in this case, SharePoint 2013) to provide tokens instead of credentials (for example, user name and password) to their data that is hosted by a given service provider (that is, SharePoint 2013).

Each token grants access to a specific site (for example, a SharePoint document repository) for specific resources (such as documents from a folder) and for a defined duration, say 30 minutes. This enables a user to grant a third-party site access to information that is stored with another service provider (in this case, SharePoint), without sharing their user name and password, and without sharing all the data that they have on SharePoint.

The OAuth protocol is used to authenticate and authorize apps and services. The OAuth protocol is used:

- To authorize requests by an app for SharePoint to access SharePoint resources on behalf of a user.
- To authenticate apps in the Office Store, an app catalog, or a developer tenant.

If the apps and services that you build for SharePoint 2013 do not require the previous authentication approach, you do not have to use the OAuth protocol when you create your apps and services.

If you plan to build an app for SharePoint that runs in a remote web app and communicates back to SharePoint 2013, you will need to use OAuth.

App Authentication

In SharePoint 2013, a typical app for SharePoint usage scenario is as follows:

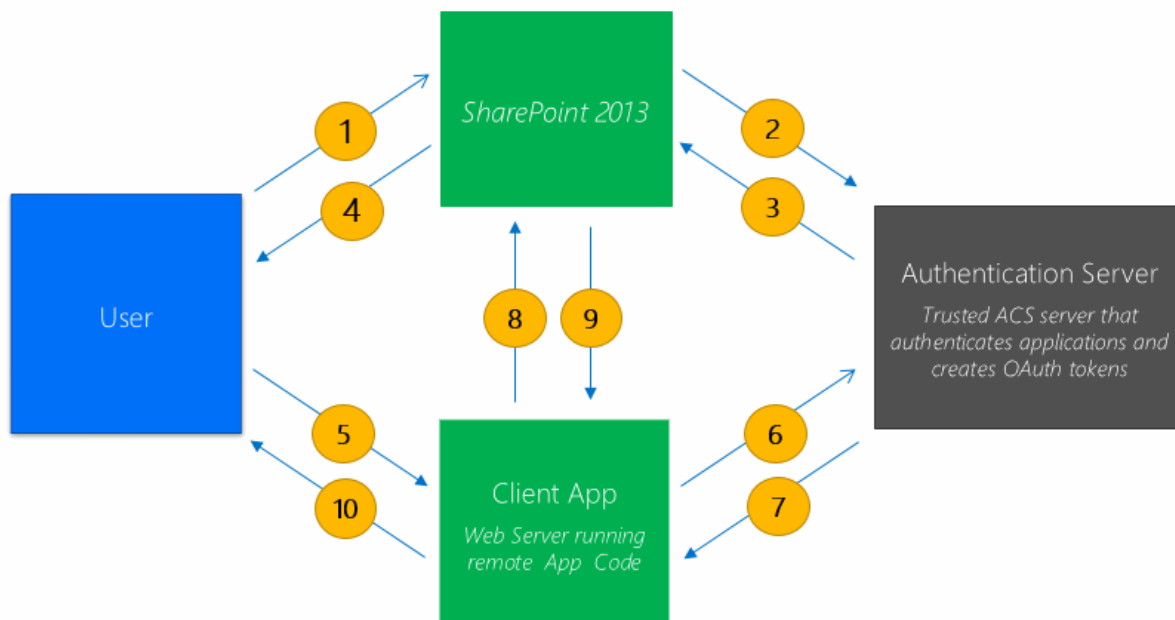


Figure 24: SharePoint authentication flow

1. User connects to the SharePoint site. SharePoint authenticates the user and creates a Security Assertion Markup Language (SAML) token which contains information about the user's identity.
2. SharePoint queries its application management database using the App ID and determines whether the app is external or not. If the app is external and has a registered app principal, SharePoint calls the Access Control System (ACS) to create a context token that passes information about the app and about the current user.
3. The ACS creates a context token which contains information about the app principal and about the user. The context token also contains a refresh token, which is used by the client app. Also note that certain aspects of the context token are signed by ACS using the app secret and can only be prepared by the client app, which also has a copy of the app secret.

4. SharePoint returns a page to the browser which contains a launcher (e.g., a clickable tile) allowing the user to redirect from the SharePoint site to the client app.
5. When the user clicks on the launcher, JavaScript behind the launcher issues an HTTP POST request to redirect the user to the client app. The body of this HTTP POST request contains the context token as a named form parameter.
6. The client app reads the context token and extracts the refresh token from inside. The client app passes the refresh token to the ACS in a request to create an OAuth token. Part of the message to ACS is signed with the app secret.
7. ACS uses the app secret to authenticate the client app request. If authentication succeeds, ACS creates an OAuth token and returns it back to the client app.
8. The client app uses the OAuth token to make CSOM calls and REST calls into SharePoint.
9. SharePoint authenticates the client app and makes sure it has the proper permissions to ensure it is authorized to do whatever it is attempting to do. If the call is authenticated and authorized, SharePoint performs whatever work is requested by the CSOM and/or REST calls, and then returns any information requested back to the client app.
10. The client app returns a page back to the user which contains HTML-generated form data returned by CSOM and REST calls into SharePoint.

App Authorization

The authorization process verifies that an authenticated subject (an app or a user the app is acting on behalf of) has permission to perform certain operations or to access specific resources (for example, a list or a SharePoint document folder).

SharePoint 2013 provides three types of authorization policies. The authenticated identities on the call determine which authorization policy is used.

The authenticated identities can be:

- User identity only
- User and app identities
- App identity only

The authorization policy can be:

- User-only policy
- User+app policy
- App-only policy

For example, if the user+app policy is used, both the app and the user the app is acting on behalf of must have permissions to the resources being accessed. In the case of app-only policies, the access check only goes against the app's rights. In Figure 25, you can analyze the security flow that the SharePoint Engine uses for user authentication.

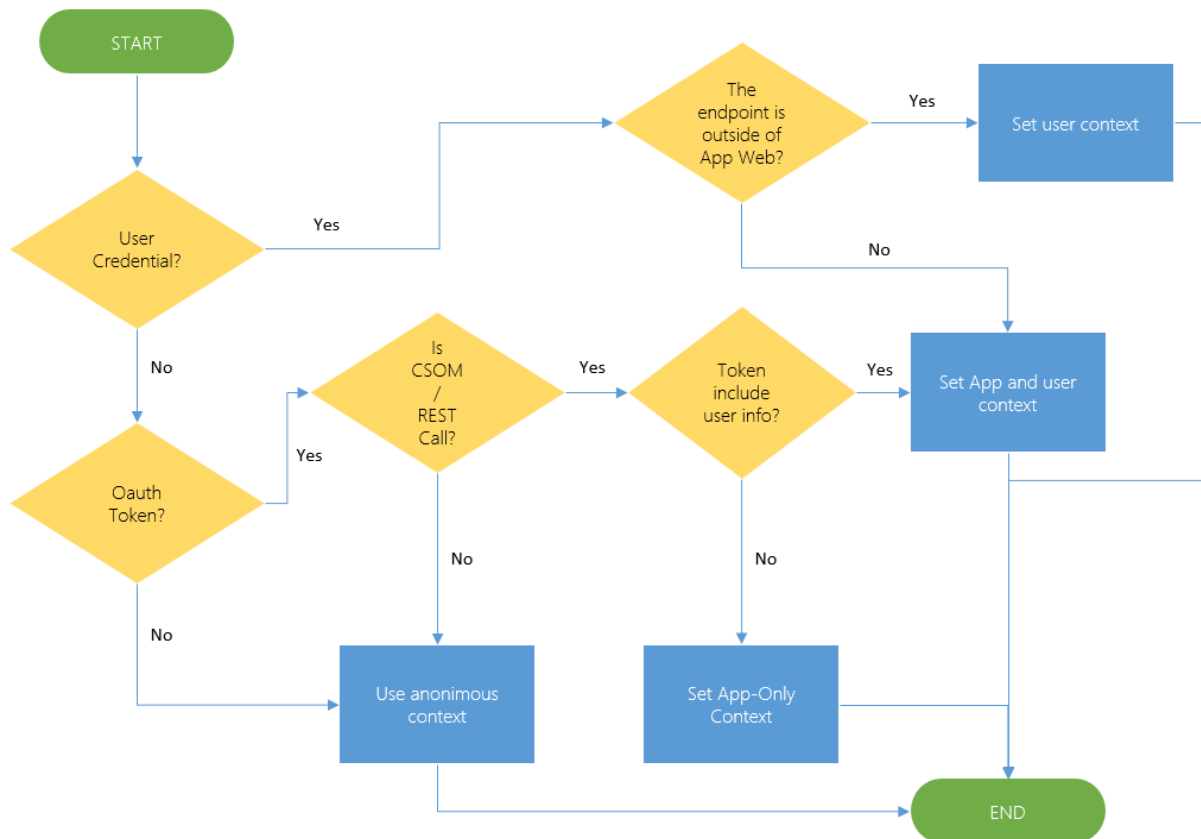


Figure 25: Security flow

The permissions that the app must have in order to work are defined within the AppManifest.xml file and are defined by three values:

1. **Scope:** Identifies the resource for which you want to set the permission.
2. **Permission:** Indicates the permission that you want to set.
3. **Properties:** Identifies additional properties needed to determine which resource the permission will be applied to (not always necessary).

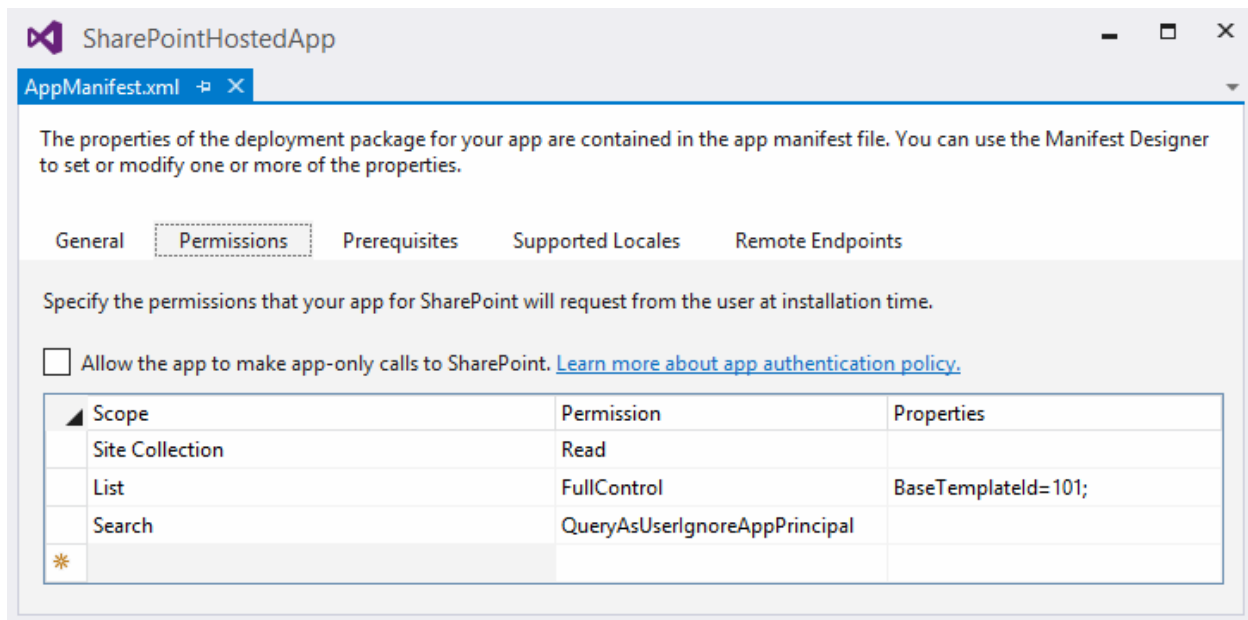


Figure 26: Set permissions on AppManifest.xml using Designer

Here are the same permissions in XML format:

```
<AppPermissionRequests>
  <AppPermissionRequest
    Scope="http://sharepoint/content/sitecollection"
    Right="Read" />
  <AppPermissionRequest
    Scope="http://sharepoint/content/sitecollection/web/list"
    Right="FullControl">
    <Property Name="BaseTemplateId" Value="101"/>
  </AppPermissionRequest>
  <AppPermissionRequest
    Scope="http://sharepoint/search"
    Right="QueryAsUserIgnoreAppPrincipal" />
</AppPermissionRequests>
```

But how do permissions work? Take a look at Figure 27.

```
<AppPermissionRequest
  Scope="http://sharepoint/content/sitecollection" Right="Read"/>
```

Diagram illustrating the components of the `Scope` attribute in the `AppPermissionRequest` XML element:

- `http`: Product
- `sharepoint`: Permission Provider
- `content/sitecollection`: Target Object
- `Read`: Capability

Figure 27: AppPermissionRequest

An app permission request contains a scope attribute and a right attribute. The scope attribute contains a string that includes the following pieces of information:

- **Product:** For example, SharePoint vs. Exchange vs. Lync
- **Permission Provider:** This is the request for permissions to content or a specific service, such as search or user profiles.
- **Target object:** An object where a grant is requested, such as a tenancy, site collection, site, or list.

The right attribute defines the type of permission being requested. SharePoint standardizes four basic rights: **Read**, **Write**, **Manage** and **FullControl**. However, some permission types, such as search permissions, use specialized permissions like **QueryAsUserIgnoreAppPrincipal**.

The flag on the page allows you to choose which type of permits the app must have:

- **User+app policy:** When the user+app policy is used, the authorization checks take into account both the user identity and the app identity. In particular, when this policy is used, authorization checks succeed only if both the current user and the app have sufficient permissions to perform the action in question.
- **App-only policy:** When the app-only policy is used, the content database authorization checks take into account only the app identity. In particular, when this policy is used, an authorization check succeeds only if the current app has sufficient permissions to perform the action in question regardless of the permissions of the current user (if any).

Summary

In this chapter, we learned about authentication and authorization in SharePoint 2013. We have also learned about app permissions; it's very important to be able to access the code of the app with resources that SharePoint offers through the CSOM or REST API.

Chapter 7 Client-Side Object Model (CSOM)

Introduction

The CSOM was introduced in SharePoint 2010 with two aims. The first is to create apps (not SharePoint 2013 apps) that might be run outside of the server where SharePoint is installed, which is the main limitation of the Server Object Model. The second is to create apps directly within SharePoint pages or web parts using its JavaScript implementation.

The main idea is that the CSOM is an object model that reflects a subset of a server-side model but with a different operation mechanism. While the server-side works directly in memory or accessing SharePoint DB, the CSOM communicates with SharePoint using a particular WCF service called Client.svc.

Internally, the Client.svc uses the Server Object Model in order to make the operations that were sent to the same service from CSOM. In the following figure, you can analyze the CSOM flow used by SharePoint which is explained in the following paragraphs:

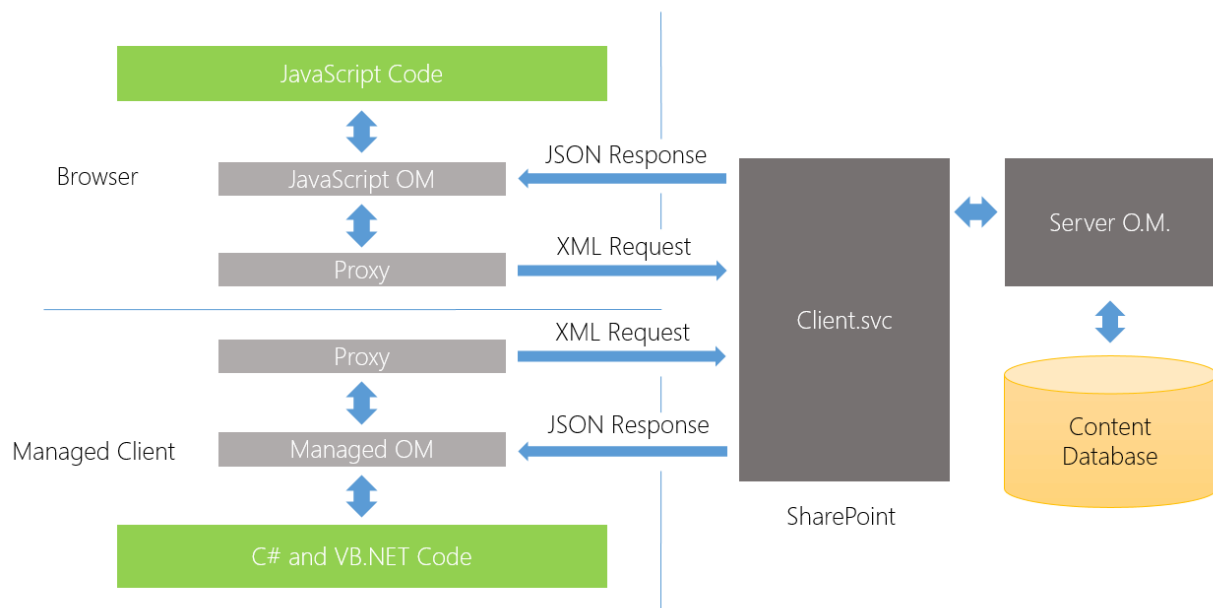


Figure 28: CSOM workflow

Regardless of its implementation, whether it is managed or JavaScript, the operating mechanism and communication with Client.svc is identical. What concerns us is how to use the CSOM.

The CSOM provides access to core SharePoint objects such as sites, lists, libraries, items, documents, etc. This version also provides access to features of the SharePoint Server such as ECM, Taxonomy, User Profiles, Search, and BCS.

The CSOM communicates with SharePoint through an object called ClientContext. This serves as a bridge between our code and SharePoint. It is the class that allows us to coordinate the sending of server operations and the subsequent information retrieval needs.

The communication between our code and SharePoint takes place in batch mode to optimize calls to the server (via the Client.svc). It means that we're going to call what we want to do, but until we make a call to **executeQuery()** or **ExecuteQueryAsync()** (asynchronous mode) of the ClientContext, no operations will actually be executed.

The idea is that the CSOM keeps track of things we want to do and, when we are ready to execute, sends them to the server side. It is there that they are parsed and executed by using the server object model, and the results are returned to the client by the objects that we used.

If we try to access properties or methods of objects of CSOM that have not been initialized by first calling the **load()** method, then the **ExecuteQuery()** method will generate an exception of the type **PropertyOrFieldNotInitializedException**.

The **load()** method loads only simple properties and not those of complex type objects or type collections. If we also want to load this property type, we must invoke the **load()** method even on those properties. This is designed to optimize the loading of objects, leaving the programmer to indicate what he or she wants to use.

The main difference between a Managed CSOM and JavaScript CSOM is the language or platform used.

In JavaScript, due to compatibility between browsers, we have properties in objects that are converted into get_xxx methods, set_xxx. The types of data that we have in JavaScript are different than the ones we have in .NET, for example.

Managed CSOM

As mentioned before, the Managed CSOM (.NET or Silverlight) offers an object model to interact with SharePoint.

Using it in an app for SharePoint, for example, a provider-hosted app created with .NET, Visual Studio automatically adds the reference in your project to two assemblies:

1. Microsoft.SharePoint.Client.dll
2. Microsoft.SharePoint.Client.Runtime.dll

There are also other DLLs to access specific features, such as:

- Microsoft.SharePoint.Client.Search.dll
- Microsoft.SharePoint.Client.Taxonomy.dll
- Microsoft.SharePoint.Client.UserProfiles.dll
- Microsoft.Office.Client.TranslationServices.dll
- Microsoft.SharePoint.Client.WorkflowServices.dll

If you wanted to use this object model for SharePoint apps, but not as an import tool or as a plug-in, you can reference these assemblies manually. They can be found in the folder **..\Web Server Extension\15\ISAPI**.

Insert the “using” directive to use the various basic objects of CSOM.

```
using Microsoft.SharePoint.Client;
```

To use the CSOM, we must create an instance of the ClientContext object so we can communicate with SharePoint. Though there are many ways to create an instance of this object, the simplest is to pass the URL of the SharePoint site. Using the code in a provider-hosted app includes a class that allows us to create a ClientContext object according to the current context, which we will discuss later.

```
var URL = "http://site/";
using (ClientContext clientContext = new ClientContext(url))
{
    // *****
    // CSOM Code goes here!
    // *****
}
```

How to Use

The following sections describe how to perform basic operations on SharePoint Web objects with a managed CSOM.

Website Tasks

The following is a series of examples on how to perform basic operations on SharePoint Web objects.

Read Properties

Here is an example of loading properties of a Web object referenced by the URL that is passed to the ClientContext that we created:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Retrieve the Web's properties.
clientContext.Load(web);

// Real execution of the query.
clientContext.ExecuteQuery();
```

This example instead lets us filter and load some properties using the lambda expression.

This allows us to load only the properties that we need or will use, which optimizes loading.

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Retrieve the Web's title and description properties.
clientContext.Load(web, x => x.Title, x => w.Description);

// Real execution of the query.
clientContext.ExecuteQuery();
```

Modify Properties

Here is an example of code that allows us to modify the properties of the SharePoint site.

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

web.Title = "Edited Title";
web.Description = "Edited Description";

// The command to update.
web.Update();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

Creating New Web

To create a new SharePoint website, we use an object (not present in the Server Object Model) that serves to inform all SharePoint values needed to create the new website.

This object is used by calling **WebCreationInformation**:

```
// Object for building a new web.
WebCreationInformation webCreation = new WebCreationInformation();

// Property setting.
webCreation.Url = "new-web";
webCreation.Title = "Welcome to the New Web";
webCreation.UseSamePermissionsAsParentSite = true;
webCreation.WebTemplate = "STS#0";

// Adding the new web to the collection of the webs.
Web web = clientContext.Web.Webs.Add(webCreation);

// Retrieve the Web's title and description properties.
clientContext.Load(web, w => w.Title, w => w.Description);

// Real execution of the query or command.
clientContext.ExecuteQuery();
```


List Tasks

The following is a series of examples on how to perform basic operations on SharePoint Lists.

Read Properties

In this example, we will use a method called **Include** that lets us indicate what to include at load time. In this case, we will load the ID and Title of all lists in the web:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Retrieve all lists.
clientContext.Load(web.Lists, lists =>
    lists.Include(list => list.Title, list => list.Id));

// Real execution of the query or command.
clientContext.ExecuteQuery();

// Enumerate the lists.
foreach (List list in web.Lists)
{
    var id = list.Id;
    var title = list.Title;
}
```

Creation of a New List

This example shows how we can create a new list on the website by using the **ListCreationInformation** object:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Object for building a new List.
ListCreationInformation creationInfo = new ListCreationInformation();

// Property setting.
creationInfo.TemplateType = (int)ListTemplateType.GenericList;
creationInfo.Title = "Generic List";
creationInfo.Description = "Description of Generic List";

// Adding the new web to the collection of webs.
List list = web.Lists.Add(creationInfo);

// Update command to set the fields as ID, etc.
list.Update();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

Add Columns to a List

Here is an example in which we can see how to add a new column to a list using the XML definition of the field. The **AddFieldByXml** method returns a base object of the type **Field**, and then we should convert it to access the properties that we want to modify:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference the list by its title.
List list = web.Lists.GetByTitle("My List");

// Adding a new field using XML syntax.
Field field = list.Fields.AddFieldAsXml(
    "<Field DisplayName='NumberField' Type='Number' />",
    true, AddFieldOptions.DefaultValue);

// Properties setting (using CastTo to cast from base object).
FieldNumber fieldNumber = clientContext.CastTo<FieldNumber>(field);
fieldNumber.MaximumValue = 90;
fieldNumber.MinimumValue = 18;

// Update command.
fieldNumber.Update();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

Deleting Columns in a List

The following is an example of how to delete a column from a list by indicating the internal name or title:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference the list by its title.
List list = web.Lists.GetByTitle("My List");

// Reference the field by its internal name or title.
Field field = list.Fields.GetByInternalNameOrTitle("Status");

// Delete command.
field.DeleteObject();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

Deleting a List

Here is an example for deleting a list:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference the list by its title.
List list = web.Lists.GetByTitle("My List");

// Command to delete.
list.DeleteObject();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

List Item Tasks

Below is a series of examples on how to perform basic operations on list items.

Query on Items

The following example shows how to query against a list using the **CamlQuery** object, allowing us to specify a query in **CAML** format (we can specify clauses, etc., in XML format):

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference the list by its title.
List list = clientContext.Web.Lists.GetByTitle("My List");

// Creates a CamlQuery that has a RowLimit of 500.
var rowLimit = 500;
CamlQuery query = CamlQuery.CreateAllItemsQuery(rowLimit);
ListItemCollection items = list.GetItems(query);

// Retrieve all items in the list.
clientContext.Load(items);

// Real execution of the query or command.
clientContext.ExecuteQuery();

foreach (ListItem listItem in items)
{
    // List item data.
    var title = listItem["Title"];
}
```

Create New Item

The following is an example of how to create a new item and add a new folder in a list:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;
```

```
// Reference the list by its title.
List list = clientContext.Web.Lists.GetByTitle("My List");

// Creating a list item.
ListItemCreationInformation listItemCreateInfo =
    new ListItemCreationInformation();

ListItem item = list.AddItem(listItemCreateInfo);
item["Title"] = "New Item";
item.Update();

// Creating a list folder.
ListItemCreationInformation folderCreateInfo =
    new ListItemCreationInformation();

folderCreateInfo.UnderlyingObjectType = FileSystemObjectType.Folder;
ListItem folder = list.AddItem(folderCreateInfo);
folder["Title"] = "New Folder";
folder.Update();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

Edit an Item

The following is an example of how to edit an item in a list by retrieving the items via its numeric ID:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference the list by its title.
List list = clientContext.Web.Lists.GetByTitle("My List");

// Get list item by ID.
var id = 1;
ListItem listItem = list.GetItemById(id);

// Write a new value to the Title field.
listItem["Title"] = "New Title";
listItem.Update();

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

Delete an Item

The following is an example of how to delete an item in a list by retrieving the items via its numeric ID:

```
// Reference to SharePoint Web.
```

```

Web web = clientContext.Web;

// Reference the list by its title.
List list = clientContext.Web.Lists.GetByTitle("My List");

// Get list item by ID.
var id = 1;
ListItem item = list.GetItemById(id);

// Delete the item.
item.DeleteObject();

// Real execution of the query or command.
clientContext.ExecuteQuery();

```

User Tasks

The following are code examples on how to work with users or groups and permissions programmatically.

Enumerate Users in a Group

The following is an example of how to do a query to find out which users belong to a SharePoint group:

```

// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference to SiteGroups of the Web.
GroupCollection groups = web.SiteGroups;

// Get group by Name.
var groupName = "Members";
Group group = groups.GetByName(groupName);

// Load Users.
clientContext.Load(group.Users);

// Real execution of the query or command.
clientContext.ExecuteQuery();

foreach (User user in group.Users)
{
    // Access to the user info.
    var title = user.Title;
}

```

Adding a User to a Group

The following is an example of how to add a user to a SharePoint group:

```

// Reference to SharePoint Web.
Web web = clientContext.Web;

// Reference to SiteGroups of the Web.
GroupCollection groups = web.SiteGroups;

// Get group by Name.
var groupName = "Members";
Group group = groups.GetByName(groupName);

// Create new user info.
UserCreationInformation userCreationInfo =
    new UserCreationInformation();
userCreationInfo.Title = "User";
userCreationInfo.LoginName = "domain\\user";
userCreationInfo.Email = "user@domain.com";

// Add the user to the group.
User newUser = group.Users.Add(userCreationInfo);

// Real execution of the query or command.
clientContext.ExecuteQuery();

```

Creating a New Role

The following is an example of how to create a new Role by setting a range of permitted:

```

// Reference to SharePoint Web.
Web web = clientContext.Web;

// Create a new Permission object with some Permissions.
BasePermissions permissions = new BasePermissions();
permissions.Set(PermissionKind.ViewListItems);
permissions.Set(PermissionKind.ViewPages);
permissions.Set(PermissionKind.ViewFormPages);

// Create a new Role Definition.
RoleDefinitionCreationInformation roleDefCreationInfo =
    new RoleDefinitionCreationInformation();
roleDefCreationInfo.BasePermissions = permissions;
roleDefCreationInfo.Name = "View Only Role";
roleDefCreationInfo.Description = "A role for View Only Users";

// Add the new Role Definition.
RoleDefinition roleDefinition =
    web.RoleDefinitions.Add(roleDefCreationInfo);

// Real execution of the query or command.
clientContext.ExecuteQuery();

```

Adding a Role to a User

The following is an example of how to add a role to a user previously created:

```
// Reference to SharePoint Web.
Web web = clientContext.Web;

// Get user by Login Name.
Principal user = web.SiteUsers.GetByLoginName(@"domain\user");

// Get Role by Name.
RoleDefinition roleDefinition =
    web.RoleDefinitions.GetByName("Read");

// Create new Role Definition.
RoleDefinitionBindingCollection roleDefinitionCollection =
    new RoleDefinitionBindingCollection(clientContext);

roleDefinitionCollection.Add(roleDefinition);
RoleAssignment roleAssignment = web.RoleAssignments.Add(
    user, roleDefinitionCollection);

// Real execution of the query or command.
clientContext.ExecuteQuery();
```

JavaScript CSOM

The JavaScript version is provided through the use of a series of .js files, including the main one called sp.js. These files should be included within the pages of your app.

These files are located in **..\web server extensions\15\TEMPLATE\LAYOUTS**.

Unlike in Managed mode where you can choose synchronous or asynchronous mode, here we are forced to use asynchronous mode.

Before we start, we need to make sure that sp.js the CSOM JavaScript, the file to use, is loaded.

If we are using a SharePoint-hosted, just check that the files are included, as shown here:

```
<script type="text/javascript"
    src="../../Scripts/jquery-1.9.1.min.js"></script>
<script type="text/javascript"
    src="/_layouts/15/sp.runtime.js"></script>
<script type="text/javascript"
    src="/_layouts/15/sp.js"></script>
```

In all other cases, we can use this technique that allows us to recover files and SP.js SP.Runtime.js from the web host:

```
<script
  src="//ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjax.js"
  type="text/javascript">
</script>
<script
  type="text/javascript"
  src="//ajax.aspnetcdn.com/ajax/jquery/jquery-1.7.2.min.js">
</script>
<script type="text/javascript">
  var hostweburl;

  // Load the required SharePoint libraries.
  $(document).ready(function () {

    // Get the URI decoded URLs.
    hostweburl =
      decodeURIComponent(
        getQueryStringParameter("SPHostUrl")
      );

    // The js files are in a URL in the form:
    // web_url/_layouts/15/resource_file
    var scriptbase = hostweburl + "/_layouts/15/";

    // Load the js files and continue to
    // the execOperation function.
    $.getScript(scriptbase + "SP.Runtime.js",
      function () {
        $.getScript(scriptbase + "SP.js", execOperation);
      }
    );
  });

  // Function to execute basic operations.
  function execOperation() {
    // Continue your program flow here.
  }

  // Function to retrieve a query string value.
  // For production purposes you may want to use
  // a library to handle the query string.
  function getQueryStringParameter(paramToRetrieve) {
    var params =
      document.URL.split("?")[1].split("&");
    var strParams = "";
    for (var i = 0; i < params.length; i = i + 1) {
      var singleParam = params[i].split("=");
      if (singleParam[0] == paramToRetrieve)
        return singleParam[1];
    }
  }
}
```



```
}  
</script>
```

How to Use

The following sections describe how to perform basic operations on SharePoint Web objects with a JavaScript CSOM.

Each function is passed as a ClientContext object which can be instantiated in different ways. For example:

```
// Using the current context.  
var clientContext = SP.ClientContext.get_current();  
  
// Using the context from the URL.  
var clientContext = new SP.ClientContext(url);
```

Website Tasks

The following is a series of examples on how to perform basic operations on SharePoint web objects.

Reading Properties

The following is an example of how to load properties of web objects referenced by the URL that is passed to the ClientContext that we created:

```
function retrieveData(clientContext) {  
    // Reference to SharePoint Web.  
    var website = clientContext.get_web();  
    // Retrieve the Web's properties.  
    clientContext.load(website);  
  
    // Real execution of the query or commands.  
    clientContext.executeQueryAsync(  
        function () {  
            var title = website.get_title();  
        },  
        function (sender, args) {  
            alert('Failed!! Error:' + args.get_message());  
        });  
}
```

For this example, let's instead filter load some properties using the lambda expression. This allows us to load only the properties we need or will use, which optimizes loading:

```
function retrieveData(clientContext) {  
    // Reference to SharePoint Web.  
    var website = clientContext.get_web();
```

```

// Retrieve the Web's properties.
clientContext.load(website, 'Title', 'Created');

// Real execution of the query or commands.
clientContext.executeQueryAsync(
    function () {
        var title = website.get_title();
        var created = website.get_created();
    },
    function (sender, args) {
        alert('Failed!! Error:' + args.get_message());
    });
}

```

Changing Properties

The following is an example of code that allows us to modify the properties of the SharePoint site:

```

function setData(clientContext) {
    // Reference to SharePoint Web.
    var website = clientContext.get_web();

    // Set the Web's properties.
    website.set_title('New Title');
    website.set_description('New Description');
    // Update
    website.update();

    // Retrieve the Web's properties.
    clientContext.load(website);

    // Real execution of the query or commands.
    clientContext.executeQueryAsync(
        function () {
            var title = website.get_title();
            var created = website.get_created();
        },
        function (sender, args) {
            alert('Failed!! Error:' + args.get_message());
        });
}

```

List Tasks

The following is a series of examples on how to perform basic operations on SharePoint Lists.

Reading Properties

In this example, we will use the load method using the string that lets us indicate what to include at load time. In this case, we will load the ID and Title of all lists in the web:

```

function retrieveData(clientContext) {
    // Reference to SharePoint Web.
    var website = clientContext.get_web();
    // Reference to SharePoint lists.
    var lists = oWebsite.get_lists();

    // Retrieve the List's properties using Include
    clientContext.load(lists, 'Include(Id, Title)');

    // Real execution of the query or commands.
    clientContext.executeQueryAsync(
        function () {
            while (collList.getEnumerator().moveNext()) {
                var list = listEnumerator.get_current();
                var id = list.get_id().toString()
                var title = list.get_title();
            }
        },
        function (sender, args) {
            alert('Failed!! Error:' + args.get_message());
        });
}

```

Creating a List

The following is an example that shows how to create a new list on the website by using the **ListCreationInformation** object:

```

function addList(clientContext) {
    // Reference to SharePoint Web.
    var website = clientContext.get_web();

    // Object for building a new list.
    var listCreationInfo = new SP.ListCreationInformation();

    // Property setting.
    listCreationInfo.set_title('New Announcements List');
    listCreationInfo.set_templateType(
        SP.ListTemplateType.announcements);

    // Adding the new list to the collection of lists.
    var list = website.get_lists().add(listCreationInfo);

    // Property setting.
    list.set_description('New Announcements List');
    // Update
    list.update();

    // Real execution of the query or commands.
    clientContext.executeQueryAsync(
        function () {
            var title = list.get_title();
        },

```

```

        function (sender, args) {
            alert('Failed!! Error:' + args.get_message());
        });
    }
}

```

Adding Columns to a List

The following example shows how to add a new column to a list using the XML definition of the Field. The **AddFieldByXml** method returns a base object of type **Field**, and then we should convert to access properties that we want to modify:

```

function addFieldToList(clientContext) {
    // Reference to list by title.
    var list = clientContext.get_web()
        .get_lists()
        .getByTitle('Announcements');

    // Add Field as XML.
    var field = list.get_fields().addFieldAsXml(
        '<Field DisplayName=\'MyField\' Type=\'Number\' />',
        true,
        SP.AddFieldOptions.defaultValue
    );

    var fieldNumber = clientContext.castTo(field, SP.FieldNumber);
    fieldNumber.set_maximumValue(100);
    fieldNumber.set_minimumValue(35);
    fieldNumber.update();

    clientContext.load(field);

    // Real execution of the query or commands.
    clientContext.executeQueryAsync(
        function () {
            var title = list.get_title();
        },
        function (sender, args) {
            alert('Failed!! Error:' + args.get_message());
        });
}

```

Deleting a List

This is a code example for how to delete a list:

```

function deleteList(clientContext) {
    // Reference to list by title.
    var list = clientContext.get_web()
        .get_lists()
        .getByTitle('Announcements');
}

```

```

// Delete list.
list.deleteObject();

// Real execution of the query or commands.
clientContext.executeQueryAsync(
    function () {
        // List deleted.
    },
    function (sender, args) {
        alert('Failed!! Error:' + args.get_message());
    });
}

```

List Item Tasks

The following is a series of examples on how to perform basic operations on list items.

Query on Items

The following is an example on how to query against a list using the **CamlQuery** object, allowing us to specify query in **CAML** format:

```

function camlQuery(clientContext) {
    // Reference to list by title.
    var list = clientContext.get_web().get_lists().getByTitle('Announcements');

    // Create CAML query.
    var camlQuery = new SP.CamlQuery();
    camlQuery.set_viewXml(
        '<View><Query><Where><Geq><FieldRef Name=\'ID\'/>' +
        '<Value Type=\'Number\'>1</Value></Geq></Where></Query>' +
        '<RowLimit>10</RowLimit></View>'
    );

    // Get items.
    var items = list.getItems(camlQuery);
    clientContext.load(items);

    // Real execution of the query or commands.
    clientContext.executeQueryAsync(
        function () {
            var listItemEnumerator = items.getEnumerator();

            while (listItemEnumerator.moveNext()) {
                var item = listItemEnumerator.get_current();
                var id = item.get_id();
                var title = item.get_item('Title');
            }
        },
        function (sender, args) {
            alert('Failed!! Error:' + args.get_message());
        });
}

```

```
}
```

Create New Item

The following is an example of how to create a new item and a new folder in a list:

```
function addItem(clientContext) {  
    // Reference to list by title.  
    var list = clientContext.get_web()  
        .get_lists()  
        .getByTitle('Announcements');  
  
    // Creating a list item.  
    var itemInfo = new SP.ListItemCreationInformation();  
    var item = list.addItem(itemInfo);  
  
    // Setting properties.  
    item.set_item('Title', 'New Item Title');  
    item.set_item('Body', 'New Item Body');  
    // Update  
    item.update();  
  
    // Real execution of the query or commands.  
    clientContext.executeQueryAsync(  
        function () {  
            var id = item.get_id();  
        },  
        function (sender, args) {  
            alert('Failed!! Error:' + args.get_message());  
        });  
}
```

Modify an Item

The following is an example of how to edit an item in a list by retrieving the item via its numeric ID:

```
function modifyItem(clientContext) {  
    // Reference to list by title.  
    var list = clientContext.get_web()  
        .get_lists()  
        .getByTitle('Announcements');  
  
    // Get Item by Numeric ID.  
    var id = 54;  
    item = oList.getItemById(id);  
    // Setting Properties  
    item.set_item('Title', 'New Title');  
    // Update  
    item.update();  
  
    // Real execution of the query or commands.
```

```

clientContext.executeQueryAsync(
    function () {
        var id = item.get_id();
    },
    function (sender, args) {
        alert('Failed!! Error:' + args.get_message());
    });
}

```

Delete an Item

The following is an example of how to delete an item in a list by retrieving the items via its numeric ID:

```

function deleteItem(clientContext) {
    // Reference to list by title.
    var list = clientContext.get_web().get_lists().getByTitle('Announcements');

    // Get item by numeric ID.
    var id = 54;
    item = oList.getItemById(id);
    // Delete the object.
    item.deleteObject();

    // Real execution of the query or commands.
    clientContext.executeQueryAsync(
        function () {
            var id = item.get_id();
        },
        function (sender, args) {
            alert('Failed!! Error:' + args.get_message());
        });
}

```

Working with the Web Host

The operations described here are acceptable when we are working on objects within the web app. When we want to access objects in the web host, we have to work with access to resources that have a different domain (remember the URL of the web app?)

We can use server-side code and OAuth to avoid cross-domain issues (because the communication is server to server), but there are potential problems when working with JavaScript.

By default, the browser locks up cross-domain calls to avoid security issues, because we could potentially have access to sensitive data to unauthorized code.

Then, when we do a call to cross-site collection, we can use the **ProxyWebRequestExecutorFactory** object (this always SP.js and SP.Runtime.js) which acts as a proxy to the SharePoint site to which we want access.

Then we use the **AppContextSite** object to reference objects that we want to use, which allows us to make invocations only if the app has access to the host site.

Below is a code example:

```
// Reference to the app context.
var context = new SP.ClientContext(appweburl)

// Reference to ProxyWebRequestExecutorFactory
var factory = new SP.ProxyWebRequestExecutorFactory(appweburl);
// Reference to the AppContextSite
var appContextSite = new SP.AppContextSite(context, hostweburl);

// Set the new executor factory.
context.set_webRequestExecutorFactory(factory);

// Use the AppContextSite to reference the objects to load.
var web = appContextSite.get_web();
// Use Context to make the load command.
context.load(web);
```

Summary

In this chapter, we learned some examples of how to use the CSOM, both Managed and JavaScript. Of course, the examples do not reflect everything that we can do with the CSOM, but serve to show the potential of this iteration model with SharePoint.

<p class="MsoNormal" style="margin-top:0in">

Chapter 8 REST/OData Service

Besides letting us access the SharePoint Client.svc service through the CSOM, SharePoint 2013 introduces the possibility of accessing it directly using REST mode, as shown in Figure 29.

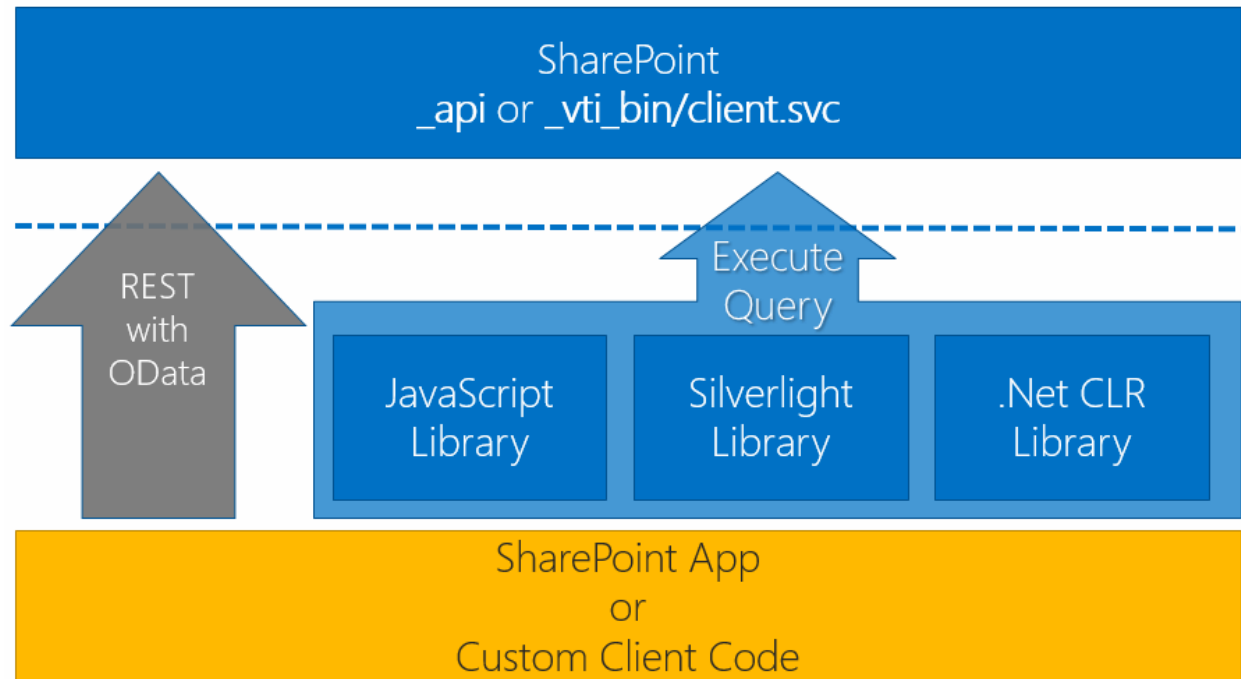


Figure 29: SharePoint client API

The REST interface gives us access to all objects and all transactions that are made available through the CSOM.

This allows us to open the SharePoint API to potentially all technologies that support the HTTP protocol and allows us to interpret strings.



Note: REST-based applications use HTTP requests to send data (create and/or update), read data (query) and delete data using HTTP verbs to make these operations (GET, POST, PUT and DELETE/MERGE).

For REST access to SharePoint 2013, we must make HTTP calls directly to the address **Client.svc** or service using the new alias **_api**.

The SharePoint REST service implements the specifications of OData to make queries with sorts, filters, etc.

Run REST Calls in SharePoint Apps

The easiest way to test a called REST in SharePoint is using the browser.

For example, if we go to this URL

https://sp2013appdevsuccinctly.sharepoint.com/_api/web/lists through the browser, we will see a result like the one shown in Figure 30:

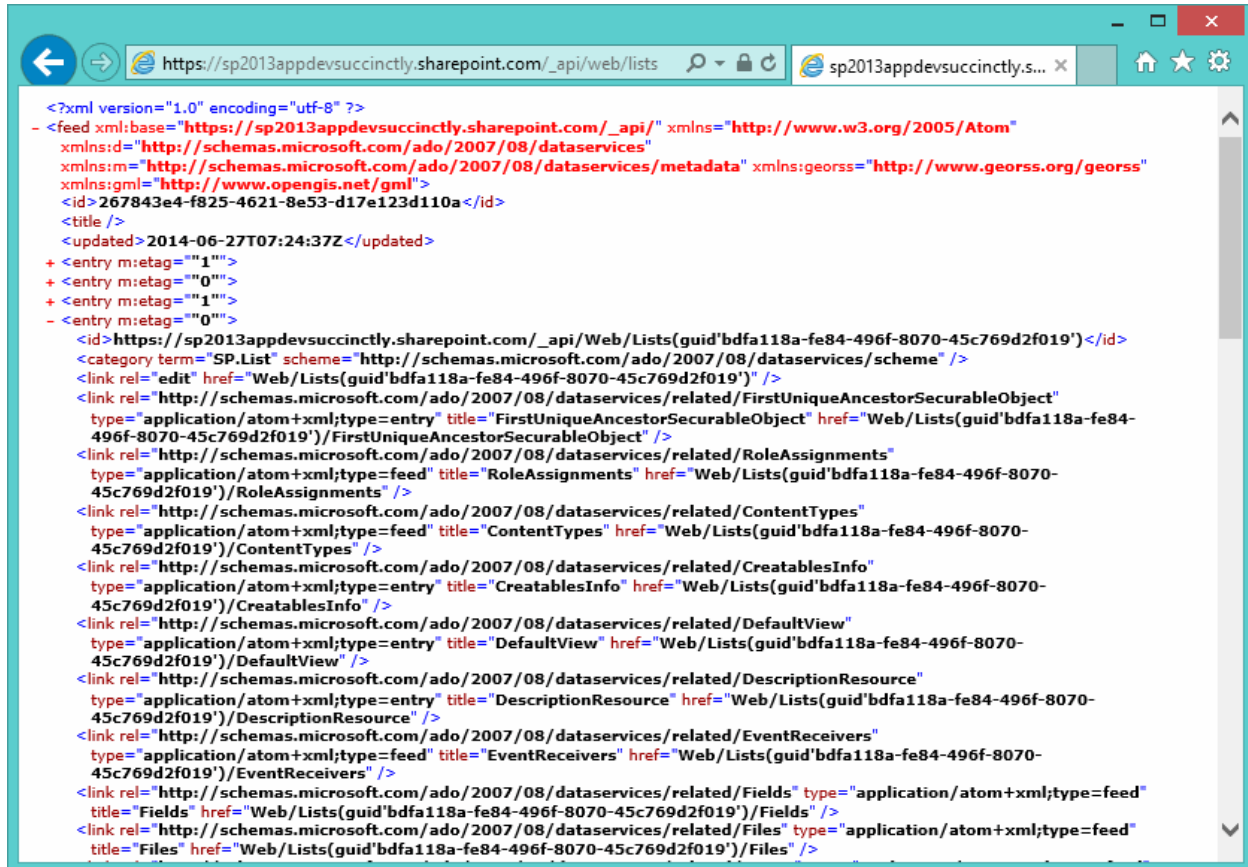


Figure 30: Call REST API from browser

Of course, we can use a browser to test a call's GET, but it is easier to use a tool like [Fiddle](#) to test our REST calls and see what data is sent and received. This is the best way to work with REST.

When we use the SharePoint REST interface, we must be careful about how we use it. There can be much difference between making calls from C# code instead of JavaScript, and it also makes a difference in the type of application that we are achieving and what type of hosting that the app will have. Furthermore, we must ensure that our app is authenticated to access data from SharePoint.

If our app is hosted on SharePoint, it then performs the client-side code in a context that allows a previously authenticated user to access it.

```
jQuery.ajax({
  URL: "http://<site URL>/_api/web/lists/GetByTitle('Test')",
  type: "GET",
  headers: {
    "accept": "application/json;odata=verbose",
    "content-type": "application/json;odata=verbose",
  },
  success: successFunction,
  error: successFunction
});
```

If the app is a self-hosted provider type, then run the server-side code (.NET or other). In this case, we should use OAuth in order to pass the access token each time we want to make a REST call to SharePoint:

```
var site = "http://site/";
var accessToken = "";
HttpRequest endpointRequest =
  (HttpRequest)HttpRequest.Create(site + "/_api/web/lists");
endpointRequest.Method = "GET";
endpointRequest.Accept = "application/json;odata=verbose";
endpointRequest.Headers
  .Add("Authorization", "Bearer " + accessToken);
HttpResponse endpointResponse =
  (HttpResponse)endpointRequest.GetResponse();
```

This is an example of C# code that does the same thing; it retrieves the lists on a SharePoint site. To make calls properly, we must also pass authentication information, passing the AccessToken authentication.

Regardless of the type of language/platform that we use, we need to understand how to create URLs that allow us to achieve the resources with which to operate.

As mentioned at the beginning of the book, the SharePoint REST API is based on the OData Protocol to define the syntax with which to create URLs. But how does OData do this?

Introduction to OData in SharePoint

OData is an HTTP-based web protocol that allows us to query and manipulate data. OData defines operations on resources using the REST paradigm and identifies those resources using a standard URI syntax. The data is transferred using AtomPub or JSON standards. Also, the OData Protocol defines some standard conventions to support the exchange of information and query schema.

Each query is submitted with a unique URL, and the results can be cached by proxy servers that increment the performance.

Table 3: OData maps CRUD operations to HTTP verbs

HTTP Verb	Operations
GET	Used to read data.
POST	Used to insert new data. Any properties that are not required are set to their default values. If you attempt to set a read-only property, the service returns an exception.
PUT and MERGE	Used for updates. MERGE is used when you want to update to some properties of an object and you want other properties to maintain their current values. PUT is used when you want to replace an item. All properties that are not specifically mentioned in the call are set with default values.
DELETE	Used to delete the object and, in the case of recyclable objects, this results in a Recyclable operation.

Moreover, the methods of CSOM will be mapped into:

- GET for Navigator operations (e.g., web.getByTitle)
- POST, PUT, MERGE or DELETE for Service operations

Take, for example, this URL:

http://server/_api/web/lists/?\$orderby=title

This URL, if invoked by calling GET, allows us to return a list of the lists in a website, sorted by title.

As all URLs in OData, this one consists of three basic parts:

- Service root URI: URL of Service (**http://server/_api**)

- Resource path: Identifies the resource on which you want to operate (**web/lists**)
- Query string options: These are the options for queries (**?\$orderby=title**)

Now take, for example, a different URL:

http://server/_api/web/lists/getByTitle('Announcements')

Let's look at how the resource is not changed in the URL (../lists), but is changed after the resource has been added. A part of the path, in this case indicating an operation to be performed, getByTitle, passes the title of the list to which you want find information.

Table 4: Examples of Resource Paths

Resource Path	Operations
web/title	Title of current site.
web/lists(guid'<list id>')	List with guid.
web/lists/getByTitle('Task')/items	The items in the Task list.
web/lists/getByTitle('Task')/fields	The fields in the Task list.
web/GetFolderByServerRelativeUrl('/Shared Documents')	The root folder of the "Shared Documents" library.
web/GetFolderByServerRelativeUrl('/Shared Documents')/Files('todo.txt')/\$value	The content of todo.txt file from the "Shared Documents" library.

By default, the data is returned as XML in AtomPub format as extended by the OData format, but you can retrieve the data in JSON format by adding the following accept header to the HTTP request: "**accept: application/json;odata=verbose**".

Through query string parameters we can apply filters, sort, and indicate the number of items to return:

Table 5: Query string options

Query String Option	Operations
\$select	Specifies which fields are included in the returned data.

Query String Option	Operations
\$filter	Specifies which members of a collection, such as the items in a list, are returned.
\$orderby	Specifies the field that's used to sort the data before it's returned.
\$top	Returns only the first <i>n</i> items of a collection or list.
\$skip	Skips the first <i>n</i> items of a collection or list and returns the rest.
\$expand	Specifies which projected fields from a joined list are returned.

Here are some examples:

- Select the column Title:
`_api/web/lists/getByTitle('MyList')/items?$select=Title`
- Filter by column Author:
`_api/web/lists/getByTitle('MyList')/items?$filter=Author eq 'Fabio Franzini'`
- Sort by the column Title ascendingly:
`_api/web/lists/getByTitle('MyList')/items?$orderby=Title asc`
- Get the first five items by jumping 10 items:
`_api/web/lists/getByTitle('MyList')/items?$top=5&$skip=10`
- Select the column Title and column Title for a LookUp, stating explicitly to expand the item LookUp:
`_api/web/lists/getByTitle('MyList')/items?$select=Title,LookUp/Title&$expand=LookUp`

So far, the examples we have seen have always taken GET calls into consideration to perform the query.

In case we want to instead insert or modify data, as well as change the HTTP verb for the request, we must pass a value called Form Digest, which can be found in two ways:

- By calling the URL **`_api/contextinfo`** always using an HTTP GET call.
- By using a hidden field called **`__REQUESTDIGEST`** contained in all SharePoint pages.

Usage Examples in JavaScript

Reading a List

```
jQuery.ajax({
  URL: "http://<site URL>/_api/web/lists/GetByTitle('Test')",
  type: "GET",
  headers: {
    "accept": "application/json;odata=verbose",
    "content-type": "application/json;odata=verbose",
  },
  success: successFunction,
  error: successFunction
});
```

Creating a New List

```
jQuery.ajax({
  URL: "http://<site URL>/_api/web/lists",
  type: "POST",
  data: JSON.stringify({ '__metadata': { 'type': 'SP.List' },
    'AllowContentTypes': true,
    'BaseTemplate': 100,
    'ContentTypesEnabled': true,
    'Description': 'My list description',
    'Title': 'Test' }),
  headers: {
    "accept": "application/json;odata=verbose",
    "content-type": "application/json;odata=verbose",
    "content-length": <length of post body>,
    "X-RequestDigest": $("#__REQUESTDIGEST").val()
  },
  success: successFunction,
  error: successFunction
});
```

Editing a List

```
jQuery.ajax({
  URL: "http://<site URL>/_api/web/lists/GetByTitle('Test')",
  type: "POST",
  data: JSON.stringify({
    '__metadata': { 'type': 'SP.List' },
    'Title': 'New title'
  }),
  headers: {
```

```

        "X-HTTP-Method": "MERGE",
        "accept": "application/json;odata=verbose",
        "content-type": "application/json;odata=verbose",
        "content-length": <length of post body>,
        "X-RequestDigest": $("#__REQUESTDIGEST").val(),
        "IF-MATCH": "*"
    },
    success: successFunction,
    error: successFunction
});

```

The value of the IF-MATCH key in the request headers is where you specify the etag value of a list or list item. This particular value applies only to lists and list items, and it is intended to help you avoid concurrency problems when you update those entities. The previous example uses an asterisk (*) for this value, and you can use that value whenever you don't have any reason to worry about concurrency issues. Otherwise, you should obtain the etag value of a list or list item by performing a GET request that retrieves the entity. The response headers of the resulting HTTP response will pass the etag as the value of the ETag key. This value is also included in the entity metadata. The following example shows the opening <entry> tag for the XML node that contains the list information. The **m:etag** property contains the etag value.

Summary

REST is much simpler to use compared to CSOM. Always use the most convenient technology for what you are doing.

If you are using .NET code, for example, because the app is provider-hosted, it will be more convenient to use the CSOM because it is a strongly typed model that is suitable to be used with managed code.

If you are writing JavaScript code, it may be more convenient, for example, to use REST because we can reuse any experience in AJAX calls with jQuery and also because the result there is already provided, using JSON in JavaScript objects.

This is not to say that REST is for use only with JavaScript and that CSOM is for use only with .NET. Rather, you should always evaluate the scopes of use before choosing a method to work with.

Chapter 9 Cross-Domain and Remote Service Calls

When we develop client-side code, we have limitations on making calls to domains different from that of our app. This is because browser security prevents cross-domain communication.

With app development for SharePoint, in reality, we find ourselves in front of two different cases: SharePoint calls that refer, for example, to different sites or HostWeb than the app, and calls to services outside SharePoint.

In both cases, we must submit to safety rules and cannot make calls to the SharePoint site if we don't have access to those sites. This also means we cannot call external resources if the app is not authorized to do so.

Cross-domain calls

In this case, to access the REST API and work with different sites from the app web, we should use the JavaScript cross-domain library (see more below):

```
var executor = new SP.RequestExecutor(appweburl);
var request = {
    URL: appweburl
        + "/_api/SP.AppContextSite(@target)/web/lists?@target='"
        + hostweburl + "'",
    method: "GET",
    headers: { "Accept": "application/json; odata=verbose" },
    success: function () {
        // Success code.
    },
    error: function () {
        // Error code.
    }
}
executor.executeAsync();
```

This example uses JavaScript to make a REST call that returns the list of lists within the SharePoint HostWeb using the cross-domain library found in SP.RequestExecutor.js (I will explain this later).

This library uses a proxy page that is hosted in an Iframe on a page of the app to enable cross-domain communication.

When the app page and SharePoint website are in different security zones, the authorization cookie cannot be sent.

If there are no authorization cookies, the library cannot load the proxy page and we cannot communicate with SharePoint.

Remote Services Calls

To construct applications, it may be useful to make calls to remote services that are perhaps not hosted by SharePoint to find information that the app needs or to send data that may be processed outside of the app itself.

For the cross-domain issues mentioned above, we are forced to use a JavaScript library that allows us to make calls outside of SharePoint or to use a web proxy that the CSOM JavaScript provides us.

For this web proxy, before actually calling to indicate it, we need check to see if the remote endpoint that we are trying to rely on is present within the file AppManifest.xls. If SharePoint will call for us, we will then return the result; otherwise, an exception will be thrown.

Let's look at an example implemented in JavaScript on how to implement this type of call.

First of all, recall that there are referencing JavaScript libraries in the CSOM that offer the web proxy feature:

```
<script type="text/javascript"
      src="/_layouts/15/sp.runtime.js"></script>

<script type="text/javascript"
      src="/_layouts/15/sp.js"></script>
```

Now implement the code to make the call. The following example makes a call to the OData service that exposes data from the sample Northwind database at:
http://services.odata.org/Northwind/Northwind.svc:

```
"use strict";

var serviceUrl =
"http://services.odata.org/Northwind/Northwind.svc/Categories";
var context = SP.ClientContext.get_current();

// Creating an instance of the SP.WebRequestInfo
// for calling the WebProxy for SharePoint.
var request = new SP.WebRequestInfo();
// Set the URL.
request.set_url(serviceUrl);
// Set the HTTP Method.
request.set_method("GET");
// Set response format.
```

```

request.set_headers({ "Accept": "application/json;odata=verbose" });
// Make command to invoke.
var response = SP.WebProxy.invoke(context, request);

// Invoke the request.
context.executeQueryAsync(successHandler, errorHandler);

// Event handler for the success event.
function successHandler() {
    if (response.get_statusCode() == 200) {
        // Load the OData source from the response.
        var categories = JSON.parse(response.get_body());

        // Extract the CategoryName and description.
        for (var i = 0; i < categories.d.results.length; i++) {
            var categoryName = categories.d.results[i].CategoryName;
            var description = categories.d.results[i].Description;
            // Make some operation on the data.
            // *****
        }
    }
    else {
        var errorMessage = "Status Code: " +
            response.get_statusCode() +
            "Message: " + response.get_body();
        alert(errorMessage);
    }
}

// Event handler for the error event.
function errorHandler() {
    var errorMessage = response.get_body();
    alert(errorMessage);
}

```

As we can see, the WebRequestInfo object is a bridge to make the call. On this subject we set the URL, the HTTP Method, and any HTTP Headers. Then, through the ClientContext we will make the call that will return the requested data to the service.

If we want to make the same call but use only a REST call, without using the CSOM, we can do so in the following way:

```

"use strict";

var serviceUrl =
    "http://services.odata.org/Northwind/Northwind.svc/Categories";

// Make a POST request to the SP.WebProxy.Invoke endpoint.
$.ajax({

```

```

URL: "/_api/SP.WebProxy.invoke",
type: "POST",
data: JSON.stringify(
    {
        "requestInfo": {
            "__metadata": { "type": "SP.WebRequestInfo" },
            "Url": serviceUrl,
            "Method": "GET",
            "Headers": {
                "results": [{
                    "__metadata": { "type": "SP.KeyValue" },
                    "Key": "Accept",
                    "Value": "application/json;odata=verbose",
                    "ValueType": "Edm.String"
                }]
            }
        },
        headers: {
            "Accept": "application/json;odata=verbose",
            "Content-Type": "application/json;odata=verbose",
            "X-RequestDigest": $("#__REQUESTDIGEST").val()
        },
        success: successHandler,
        error: errorHandler
    });

// Event handler for the success event.
function successHandler(data) {
    // Check for status code == 200
    // Some other status codes, such as 302 redirect,
    // do not trigger the errorHandler.
    if (data.d.Invoke.StatusCode == 200) {
        var categories = JSON.parse(data.d.Invoke.Body);
        // Extract the CategoryName and Description
        for (var i = 0; i < categories.d.results.length; i++) {
            var categoryName = categories.d.results[i].CategoryName;
            var description = categories.d.results[i].Description;
            // Make some operation on the data.
            // *****
        }
    }
    else {
        alert("Error!");
    }
}

// Event handler for the success event.
function errorHandler() {

```

```
    alert("Error!");  
}
```

In this case, we see how a call is made using the jQuery AJAX method, passing a series of parameters required to inform the REST service `_api` that you're trying to make a call to a remote endpoint.

You have the choice to use the CSOM or REST mode to make a call to a remote endpoint.

Summary

In this chapter, we learned about the problems of cross-domain communication that exist in SharePoint, and we have learned how to work around them using the capabilities that are provided by CSOM and the REST API.

Chapter 10 Creating User Experience (UX) for Apps

When creating an app for SharePoint, both the look and feel of the app play a fundamental role to its success with users. We need to ask ourselves if the look and feel of our app is consistent with the look and feel of SharePoint; is the experience of using our app consistent with respect to the standard mechanisms of SharePoint?

Look and Feel

Again, the look and feel of an app is very important. If our app has a look and feel that's consistent with SharePoint, it will provide a better user experience.

Using Default Styles

You can refer to the style sheet of a SharePoint site in the SharePoint app. The interesting thing is that if you change the theme or style sheet of your SharePoint site, it automatically applies the new style sheet in your app without changing the style reference sheet in your app.

This will inherit all styles, and you can use all the classes and selectors present within the SharePoint style sheet, and then apply a style consistent with the SharePoint site.

H1 with ms-core-pageTitle class

H1 with ms-accentText class

H2 with ms-accentText class

H2 with ms-webpart-titleText class

ANCHOR WITH MS-COMMANDLINK CLASS

This sample shows you how to use some of the classes defined in the SharePoint website's style sheet.

Figure 31: Using default styles

```
<h1 class="ms-core-pageTitle">H1 with ms-core-pageTitle class</h1>
<h1 class="ms-accentText">H1 with ms-accentText class</h1>
<h2 class="ms-accentText">H2 with ms-accentText class</h2>
<div>
  <h2 class="ms-webpart-titleText">H2 with ms-webpart-titleText
    class</h2>
```

```

<a class="ms-commandLink" href="#">Anchor with ms-commandLink
  class</a>
<br />
<p>
  This sample shows you how to use some of the classes defined
  in the SharePoint website's style sheet.
</p>
</div>

```

There are basically three different ways to reference a SharePoint style sheet, depending on what kind of app we're developing:

- **Use the default master page of the app:** In this case, we can use this technique only if we are making SharePoint-hosted apps because the master page is already the reference to the style sheet of your SharePoint site.
- **Reference and use the chrome control:** In this case, we can use a client-side control, provided by SharePoint, which allows us to apply in semi-automatic mode a layout similar to that of the default master page of SharePoint. It is used usually when you realize the app-hosted provider because they cannot directly use the SharePoint master pages. We will explain further details about this mode later.
- **Reference the style sheet manually:** You can manually reference the style sheet file in all cases, such as when you are making a page and don't want to use the master page of the SharePoint chrome control.

We can reference the style sheet manually by adding a tag link and creating a href that contains the URL of the web host, retrieving it dynamically. This is great because we want to render the tag's link directly on the server side, placing it on the page before use in the body:

```

<link rel="stylesheet"
      href="{host web URL}/_layouts/15/defaultcss.ashx" />

```

Alternatively, we can dynamically load the client-side tags link, always referring to the web host, as follows:

```

(function () {
  var hostWebUrl;
  var ctag;

  // Get the URI decoded app web URL.
  var hostWebUrl = decodeURIComponent(
    getQueryStringParameter("SPHostUrl"));
  // Get the ctag from the SPClientTag token.
  var ctag = decodeURIComponent(
    getQueryStringParameter("SPClientTag"));
  // Create defaultcss.ashx URL.
  var defaultCssUrl = hostWebUrl +
    "/_layouts/15/defaultcss.ashx?ctag=" + ctag;

  // Dynamically create the link element.
  var link = document.createElement("link");
  link.setAttribute("rel", "stylesheet");
  link.setAttribute("href", defaultCssUrl);

```

```

// Append the link element in the document head.
var head = document.getElementsByTagName("head");
head[0].appendChild(link);
})();

// Function to retrieve a query string value
function getQueryStringParameter(paramToRetrieve) {
    var params;
    var strParams;

    params = document.URL.split("?")[1].split("&");
    strParams = "";
    for (var i = 0; i < params.length; i = i + 1) {
        var singleParam = params[i].split("=");
        if (singleParam[0] == paramToRetrieve)
            return singleParam[1];
    }
}

```

The only problem with this method occurs when the link is loaded within the page. Since the HTML of the page displays what the CSS will actually be linked to, the page will be a light flash on the page because styles are applied only when the CSS is actually loaded. One technique to fix this would be to hide the body of the HTML page and have it display only when the loading of CSS is done.

Using the Chrome Control

As mentioned before, this technique allows you to create a layout that is very similar to that of SharePoint without using the master page.

It's very convenient to use the chrome control to create HTML pages for the SharePoint-hosted applications or for the provider-hosted applications that can not use the master page in SharePoint in any way.

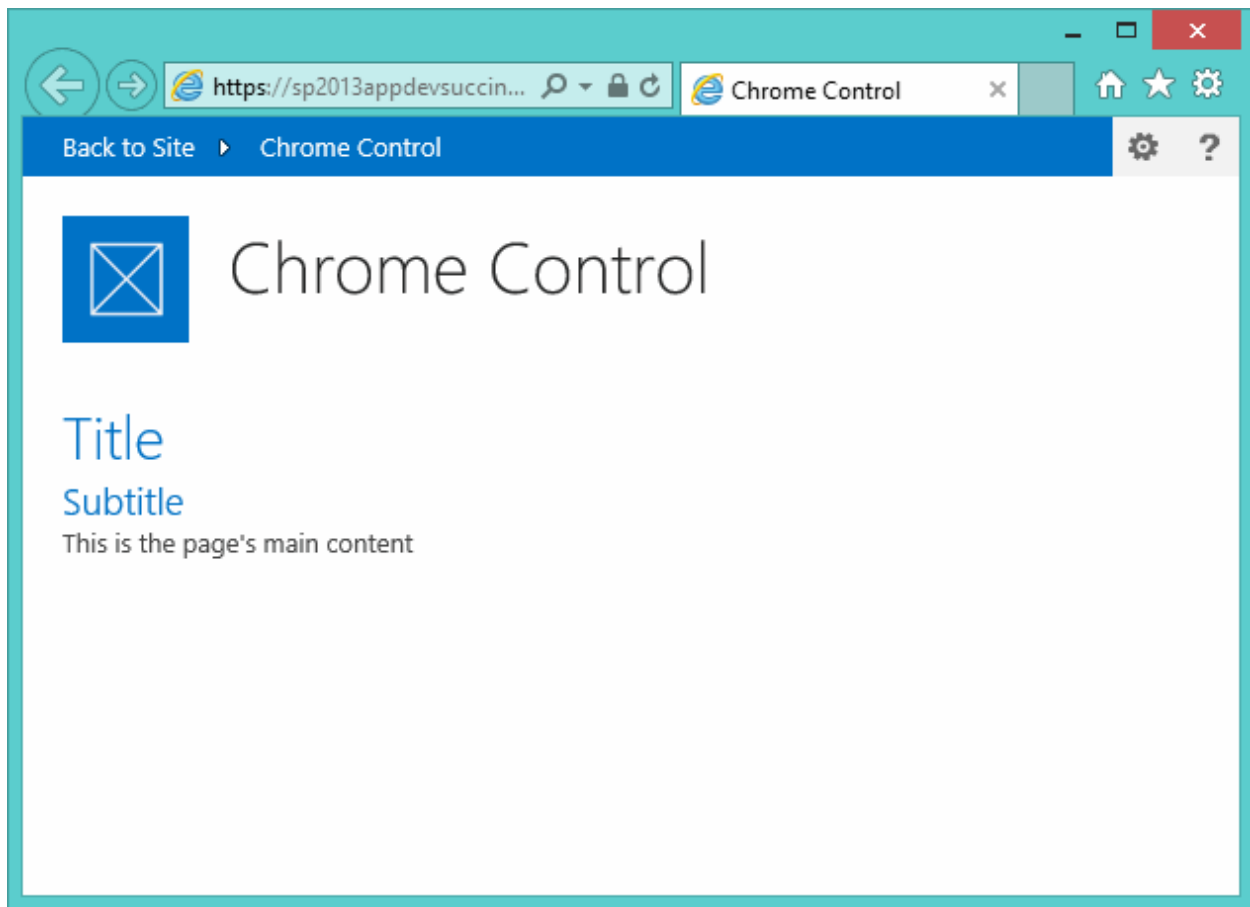


Figure 32: Chrome control in action

The chrome control contains a bar at the top of the page that includes a link called “Back to Site,” which lets you return to the web host. On the right, the Settings menu and Help button are visible only because they were passed to the configuration.

Another advantage is that we are automatically linked to the style of our SharePoint site, allowing us to use all the basic styles.

To use it we need to write JavaScript code that will be executed by the client-side page load.

We need to reference a file called **SP.UI.Controls.js** in SharePoint and then write code to initialize it.

The following is hypothetical body content to put into a page:

```
<body style="display: none">
  <!-- Chrome control placeholder -->
  <div id="chrome_ctrl_placeholder"></div>
  <div style="margin: 0px 20px 0px 20px">
    <!-- The chrome control also makes the SharePoint Website -->
    <!-- style sheet available to your page. -->
    <h1 class="ms-accentText">Title</h1>
```

```

    <h2 class="ms-accentText">Subtitle</h2>
    <div id="MainContent">
        This is the page's main content.
    </div>
</div>
</body>

```

The div with the ID **chrome_ctrl_placeholder** serves as a placeholder to hold the HTML rendering of the chrome control.

To load and initialize the chrome control, we can write some JavaScript like this:

```

<script src="//ajax.aspnetcdn.com/ajax/4.0/1/MicrosoftAjax.js"
    type="text/javascript"></script>
<script src="../../Scripts/jquery-1.9.1.min.js"></script>
<script type="text/javascript">
    "use strict";

    $(document).ready(function () {
        //Get the URI decoded URL.
        var hostweburl = decodeURIComponent(
            getQueryStringParameter("SPHostUrl"));
        var scriptbase = hostweburl + "/_layouts/15/";

        // Load the js file and continue to renderChrome function
        $.getScript(scriptbase + "SP.UI.Controls.js", renderChrome)
    });

    //Function to prepare the options and render the control
    function renderChrome() {
        // Create options object to initialize Chrome control
        var options = {
            "appIconUrl": "../../Images/AppIcon.png",
            "appTitle": "Chrome Control",
            "appHelpPageUrl": "help.html?"
                + document.URL.split("?")[1],
            "onCssLoaded": "chromeLoaded()",
            "settingsLinks": [
                {
                    "linkUrl": "link1.html?"
                        + document.URL.split("?")[1],
                    "displayName": "Link 1"
                },
                {
                    "linkUrl": "link2.html?"
                        + document.URL.split("?")[1],
                    "displayName": "Link 2"
                }
            ]
        };

        // Create new chrome control.
        var nav = new SP.UI.Controls.Navigation(

```

```

        "chrome_ctrl_placeholder", options);
    // Show chrome control
    nav.setVisible(true);
}

// Callback for the onCssLoaded event.
function chromeLoaded() {
    // When the chrome control has loaded, display the page body.
    $("body").show();
}

// Function to retrieve a query string value.
function getQueryStringParameter(paramToRetrieve) {
    var params =
        document.URL.split("?")[1].split("&");
    var strParams = "";
    for (var i = 0; i < params.length; i = i + 1) {
        var singleParam = params[i].split("=");
        if (singleParam[0] == paramToRetrieve)
            return singleParam[1];
    }
}
</script>

```

This example involves dynamic loading, initialization script and then, when we automatically load the SharePoint style sheet, it will be made visible to the body of the page.

App Shapes

When creating apps, we can choose several ways in which the user will use the app itself.

In total, there are three modes:

- Immersive Full Page
- Client Web Part (App Part)
- Custom Action

Immersive Full Page

Immersive Full Page is the default mode; each app must contain at least one page that is also the main page. When you click on the app, SharePoint redirects the request to the main page of the app, which is actually the entry point of the app.

Client Web Part

This mode allows you to implement the concept of web parts by using the apps. It is also called an App Part. Once the app is installed, if there are parts of the app, the user can choose them and insert them inside your SharePoint pages by selecting them from the catalog of app parts.

It is simple to create an app part. Visual Studio provides a project item called a Web Part Client.

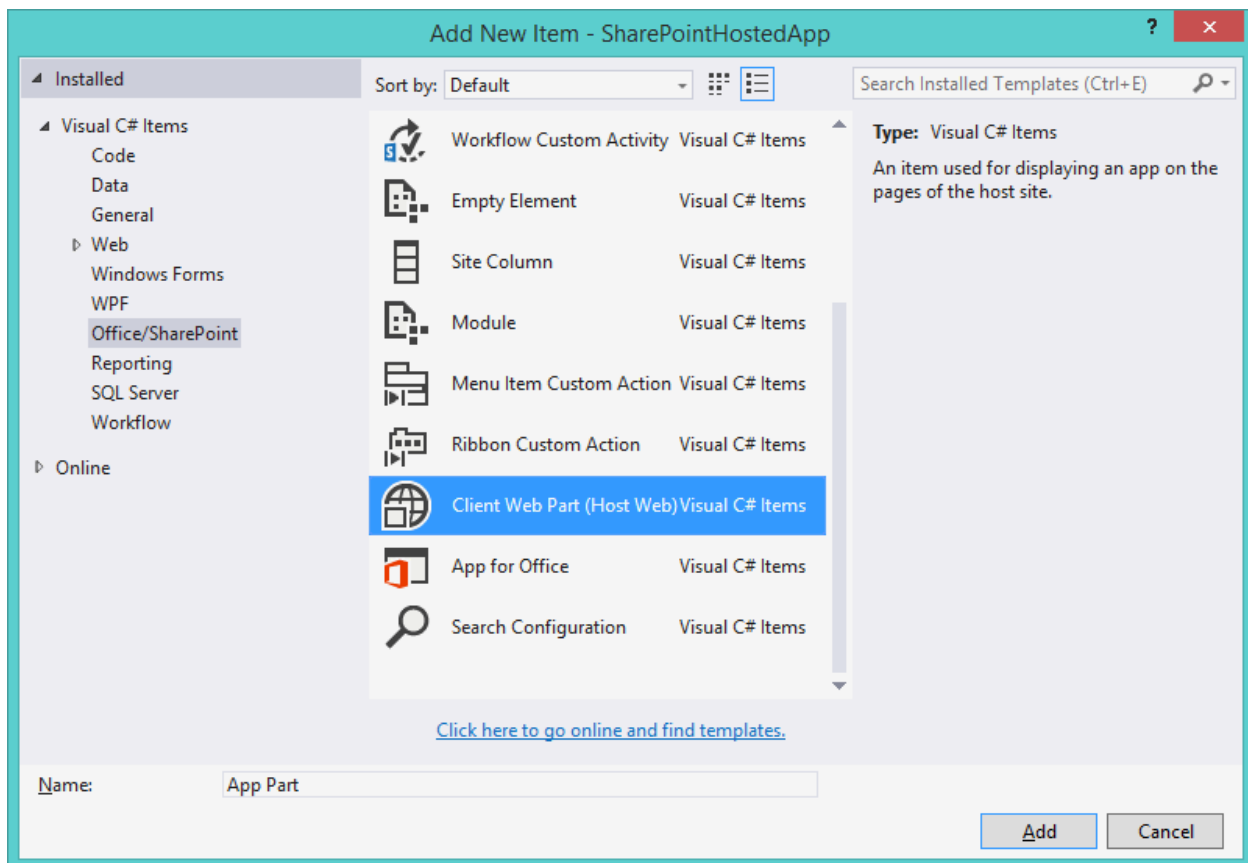


Figure 33: Add client web part

At this point, Visual Studio will prompt you to ask if you want to create a new page or select a previously created page:

Create Client Web Part

S Specify the client web part page

Client web parts display the contents of a specified web page on the host web where the app is installed. Select the option to create a new web page or specify an existing web page.

☒ **Create a new app web page for the client web part content**
A new page is created in the app for SharePoint project.

Page name:
AppPart

☐ **Select or enter the URL of an existing web page for the client web part content**
~appWebUrl/Pages/Default.aspx?{StandardTokens}

< Previous Next > **Finish** Cancel

Figure 34: Create new client web part page

This is because an app part is an HTML page that will be displayed in an IFrame on the page that will host it.

Once added to the project, we can see that the pages have been created under the module "Pages" and a new Item of the project that contains the necessary configuration for the app part has been added.

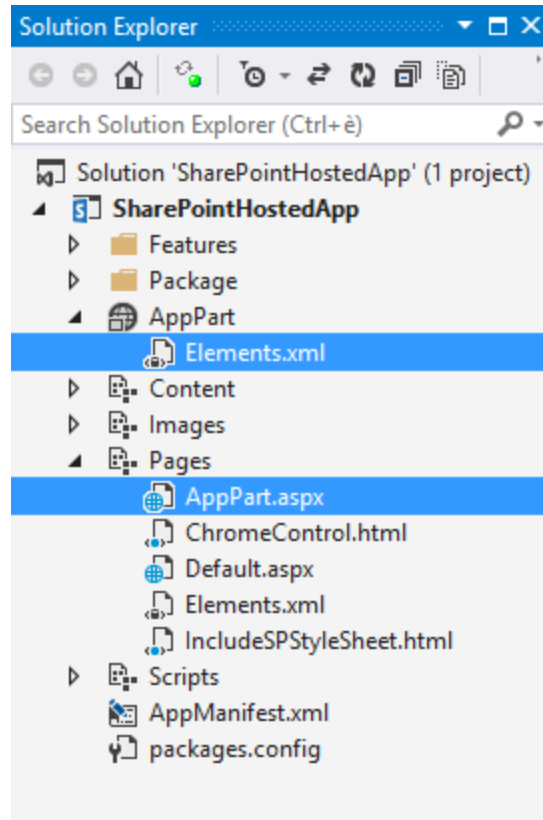


Figure 35: App part files

Inside the Element.xml, we can configure our app part in terms of Name, Title, Description, Size, and Properties that we can create. This allows users to configure the app parts as we could with the classic web part.

The properties are passed to the page through the query string using a convention that puts the name of the property in the URL of the page, with a "_" before and after the name.

For example, if we want to create a property of type string with name the "Property1", what we write is:

```
<?xml version="1.0" encoding="utf-8"?>
<Elements xmlns="http://schemas.microsoft.com/sharepoint/">
  <ClientWebPart Name="AppPart"
    Title="AppPart Title"
    Description="AppPart Description"
    DefaultWidth="300"
    DefaultHeight="200">

    <!-- Content element identifies the location of the page that
         will render inside the client web part.
         Properties are referenced on the query string using the
         pattern _propertyName_
         Example: Src=~appWebUrl/Pages/ClientWebPart1.aspx
```

```

        ?Property1=_property1_" -->
<Content Type="html" Src=~appWebUrl/Pages/AppPart.aspx
    ?{StandardTokens}&_property1_" />

<!-- Define properties in the Properties element.
    Remember to put Property Name on the Src attribute of the
    Content element above. -->
<Properties>
    <Property Name="Property1"
        Type="string"
        DefaultValue="Default Value"
        PersonalizationScope="shared"
        RequiresDesignerPermission="false"
        WebBrowsable="true"
        WebDisplayName="Property 1"
        WebCategory="Custom Property"
        WebDescription="Custom Property called Property 1"/>
</Properties>

</ClientWebPart>
</Elements>

```

We can create the following property types:

- String
- Int
- Boolean
- Enum

In order to retrieve the value of the property, just go to the page to retrieve the corresponding value in the query string.

In this example, we have created an app part in a design type that's SharePoint-hosted, but the same example applies in a project that's provider-hosted. In this case, the page was not local and was a remote web app.

The page that Visual Studio creates is an .aspx page that contains a special SharePoint control:

```

<WebPartPages:AllowFraming ID="AllowFraming" runat="server" />

```

This control enables SharePoint to host this page within an IFrame by eliminating the HTTP header X-Frame-Options from the page response. Otherwise, it would not be possible to use this page in an IFrame.

On the same page, we see that there is JavaScript that allows dynamic loading of a corev15.css file to incorporate SharePoint foundation styles:

```

if (hostUrl == '') {
    document.write('<link rel="stylesheet"
        href="/_layouts/15/1033/styles/themable/corev15.css" />');
}

```

Once the app is deployed, we can add our app part by editing the page and then selecting the app part from the catalog:

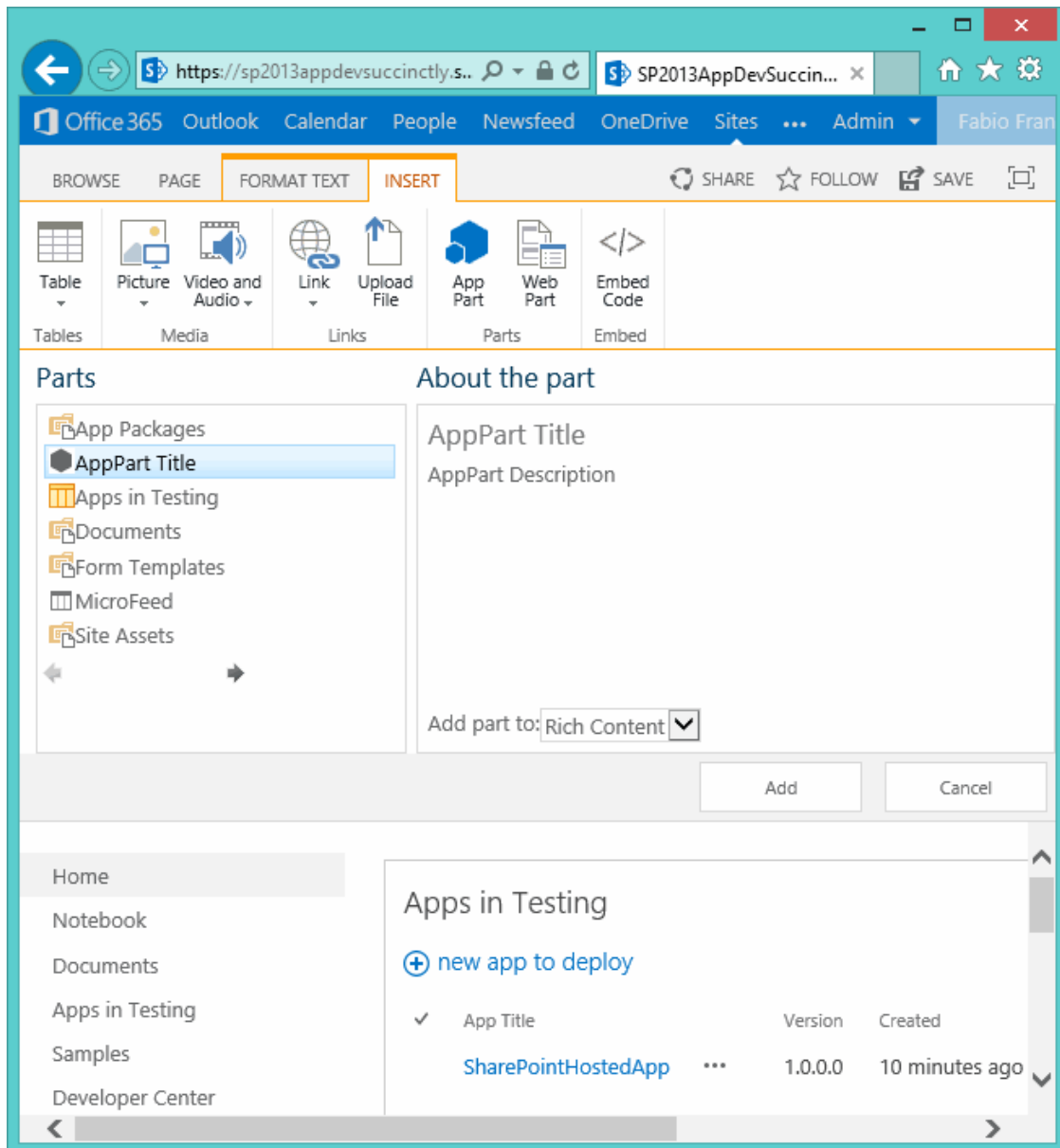


Figure 36: Insert App Part

Once you have selected and inserted into the page (previously put into edit), our app part will show the following:

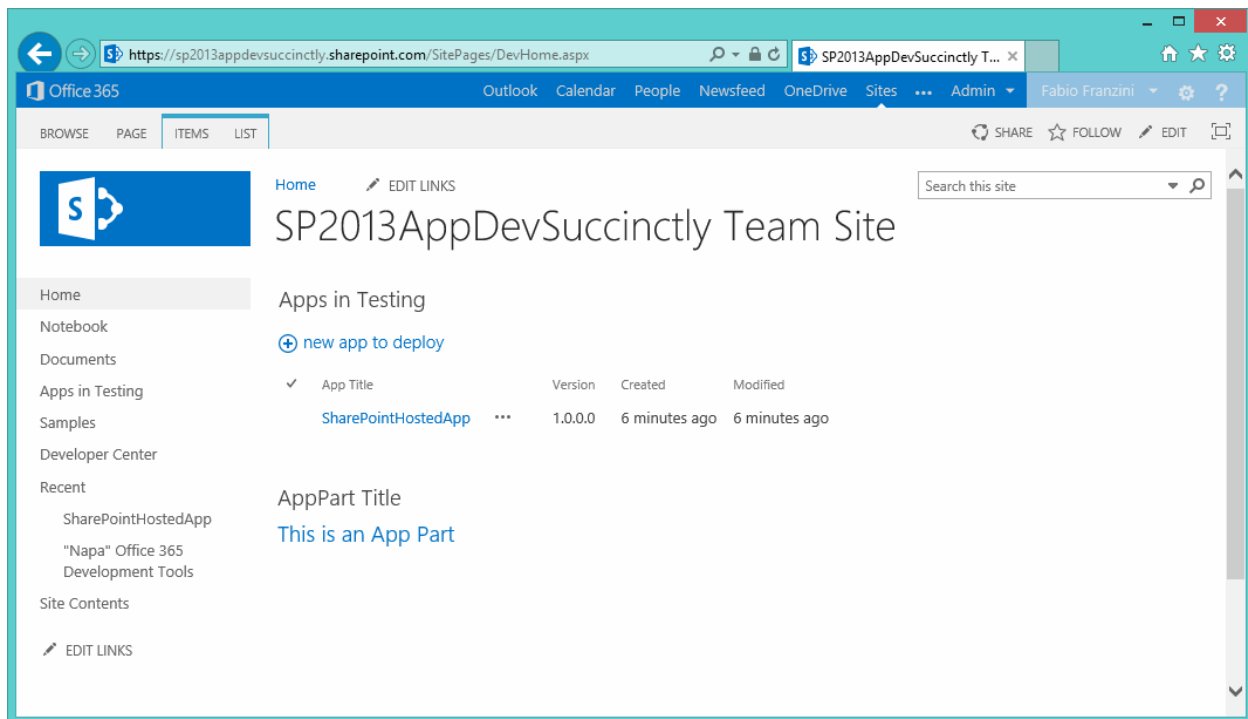


Figure 37: Example of app part inserted on page

If we need to resize the app part to fit its contents, you must use a mechanism like this:

```
function Resize() {
    var width = 500;
    var height = 300;

    var senderId = null;
    var hostUrl = null;

    if (window.parent == null)
        return;

    // Extracts the host URL and sender ID.
    var params = document.URL.split("?")[1].split("&");
    for (var i = 0; i < params.length; i = i + 1) {
        var param = params[i].split("=");
        if (hostUrl == null) {
            hostUrl = decodeURIComponent(param[1]);
        }

        if (i == (params.length - 1))
            senderId = decodeURIComponent(param[1]);
    }
    // *****

    // Use the postMessage api to resize the App part.
}
```

```

var message = "<Message senderId="
              + senderId + " >"
              + "resize(" + widthSelected
              + "," + height
              + ")</Message>";
window.parent.postMessage(message, hostUrl);
}

```

By invoking the function `resize()`, basically it lets us extract the necessary parameters from the query string, and then we can use the `postMessage` API to send a message that is formatted in this manner:

```
<message senderId={SenderId}>resize({width}, {height})</message>
```

Therefore, you can inform the host page to resize with the specified size.

Custom Action

Another type of interaction you can have with an app is the custom action. There are two types of custom actions available to the app:

1. Ribbon custom action
2. Menu Item custom action

In the first case, we can insert a button on the ribbon bar with a range of possible configurations. In the second case, we can insert a custom action within the context of the menu item within the list:

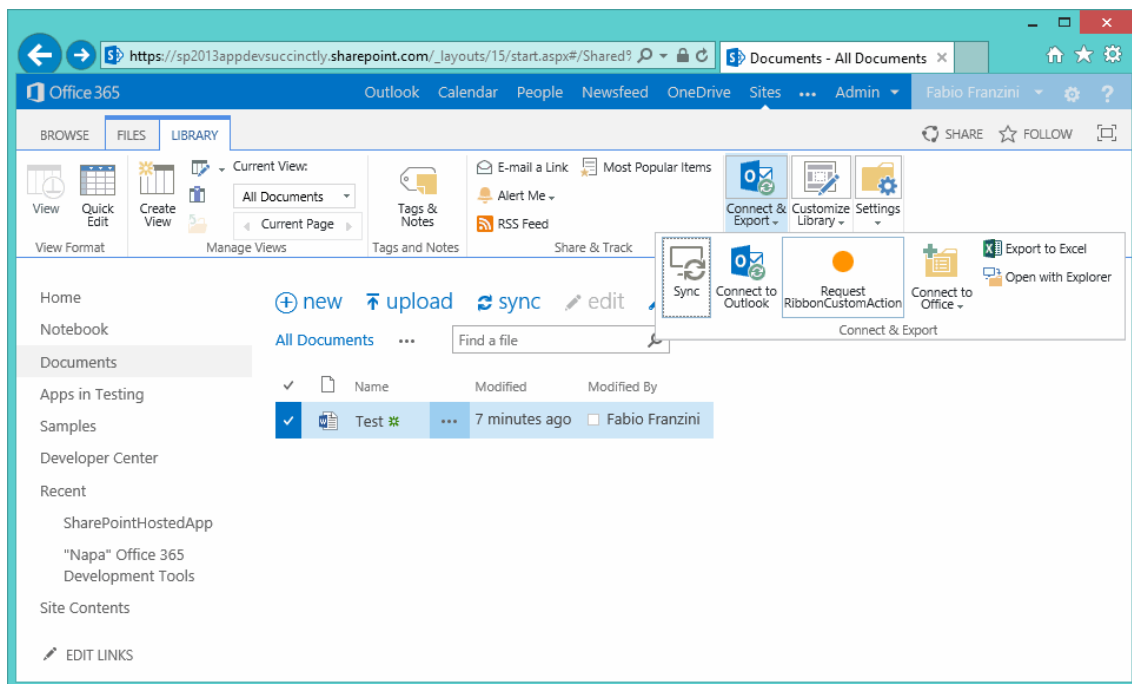


Figure 38: Custom ribbon action

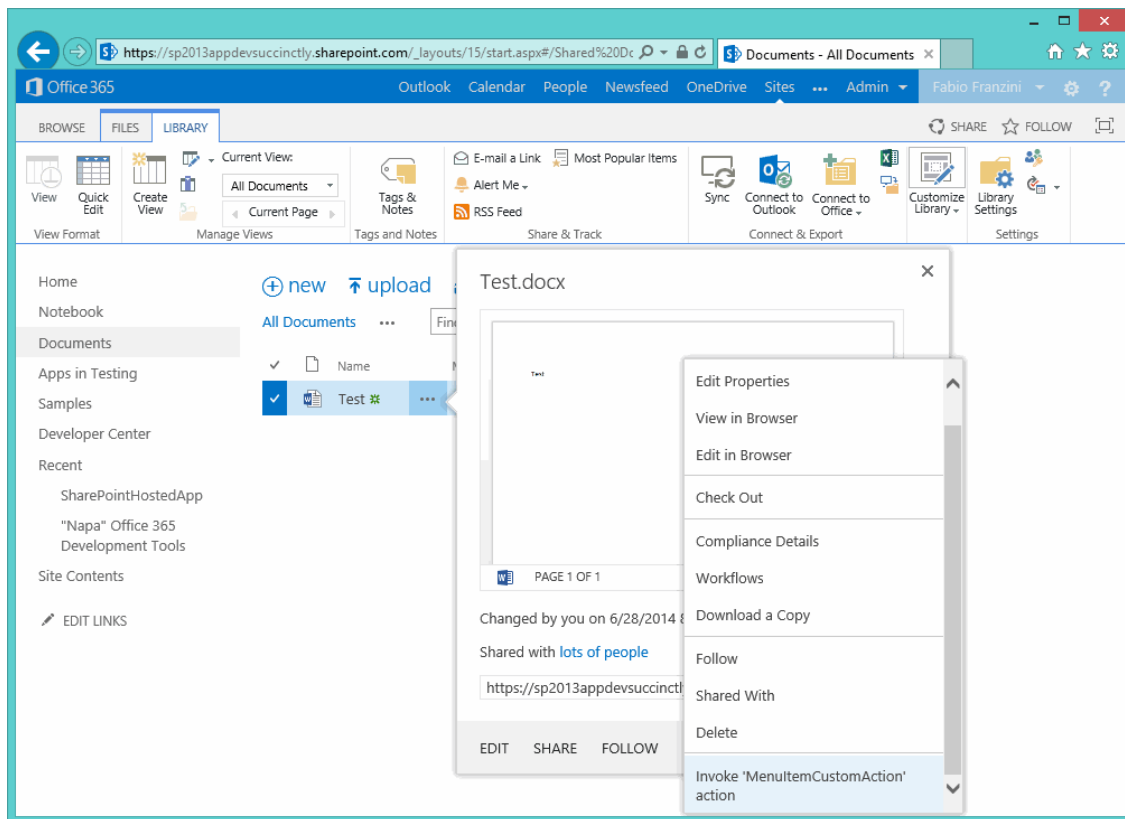


Figure 39: Menu item custom action

We can add a custom action and Ribbon List Item custom action thanks to available project templates shown in Figure 40:

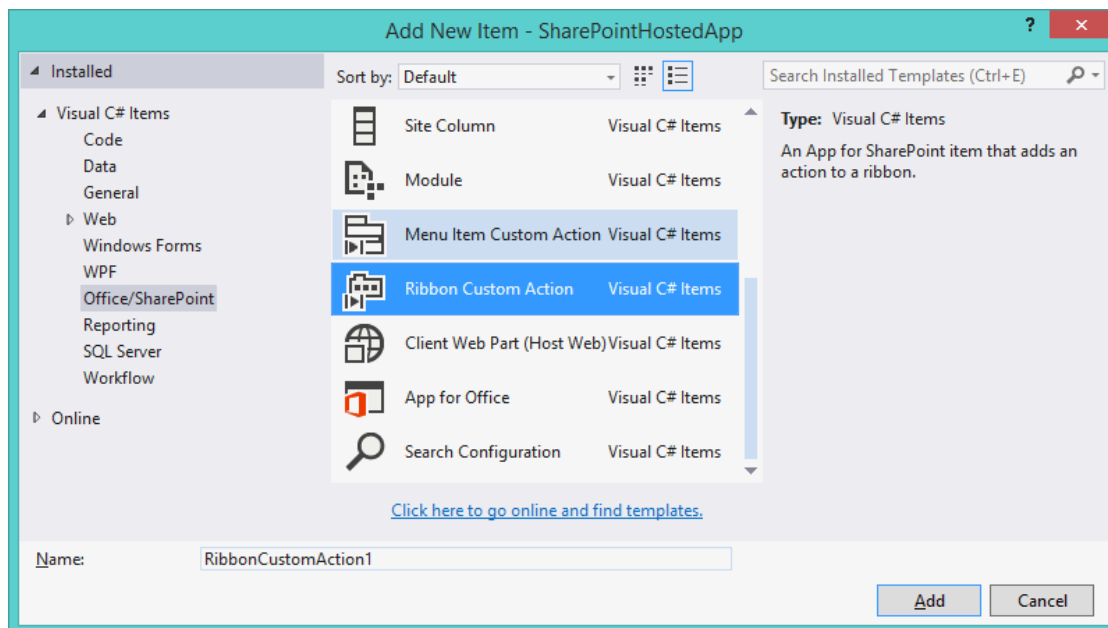


Figure 40: Add custom action

In both cases, we have to configure the kind of site that includes the custom item we insert, or a set of fundamental parameters, such as the label and landing page. In the case of a custom Ribbon action, the location of within the Ribbon control can be specified as shown in Figure 41.

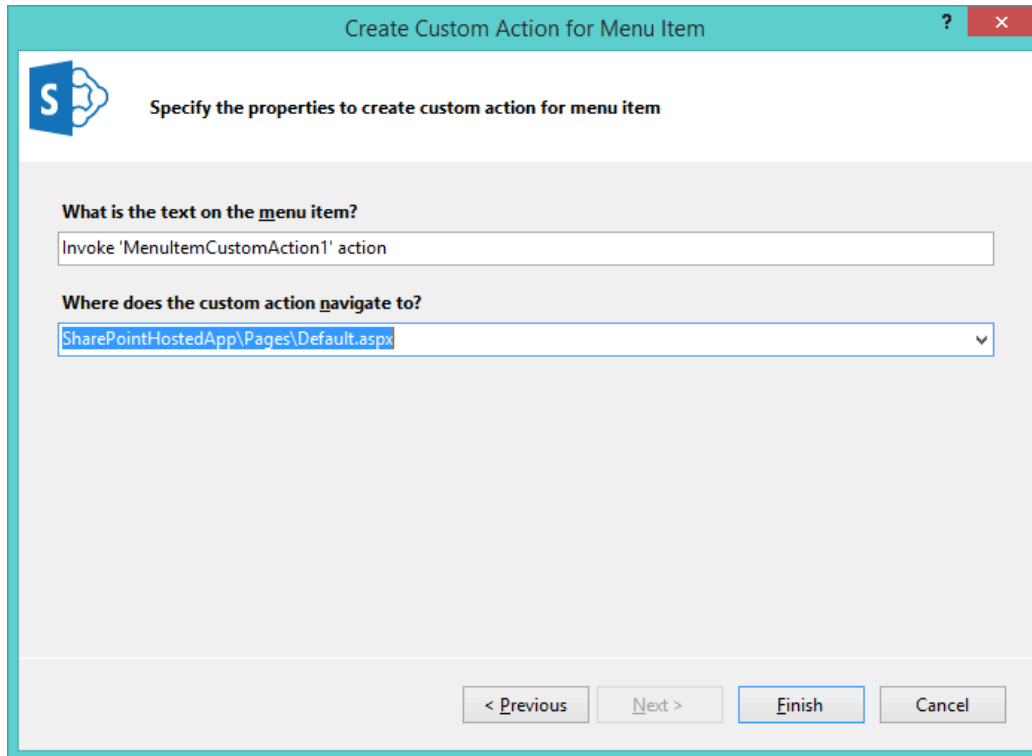


Figure 41 shows the 'Create Custom Action for Menu Item' dialog box. The dialog is titled 'Create Custom Action for Menu Item' and contains the following fields and buttons:

- What is the text on the menu item?**: A text box containing 'Invoke \'MenuItemCustomAction1\' action'.
- Where does the custom action navigate to?**: A dropdown menu showing 'SharePointHostedApp\Pages\Default.aspx'.
- Buttons**: '< Previous', 'Next >', 'Finish', and 'Cancel'.

Figure 41: Menu item custom action settings

Figure 42: Ribbon custom action settings

Every custom action creates the destination URL that will be the core page, plus a series of query string parameters like the following example:

```
?HostUrl={HostUrl}&Source={Source}&ListUrlDir={ListUrlDir}&ListID={ListID}&ItemURL={ItemUrl}&ItemID={ItemId}
```

In this way, in the target page, we will be able to recover through query string parameters like HostWeb, list ID, Items selected IDs, etc.

Summary

In this chapter, we learned how to architect our app from the UX point of view, whether we are creating a SharePoint-hosted or a provider-hosted app.

We learned that the use of the app part promotes the integration of our app in the SharePoint site where it will be used, while the use of custom action allows us to insert an extension point list level or library that is managed by the app itself.

Chapter 11 App Deployment

Introduction

Once we create our SharePoint apps, we need to decide how to publish them to make them available for installation.

We have two options, depending on whether or not we want to make our app public. We can use a private store called the App Catalog, or use one created by Microsoft called the Office Store—which all users of SharePoint can access via the Internet.

In either case, we need to generate the file .app through the publishing process that Visual Studio provides.

Publishing

The first thing to do is to generate the app through the publishing process that Visual Studio provides in the project, as shown in Figure 43.

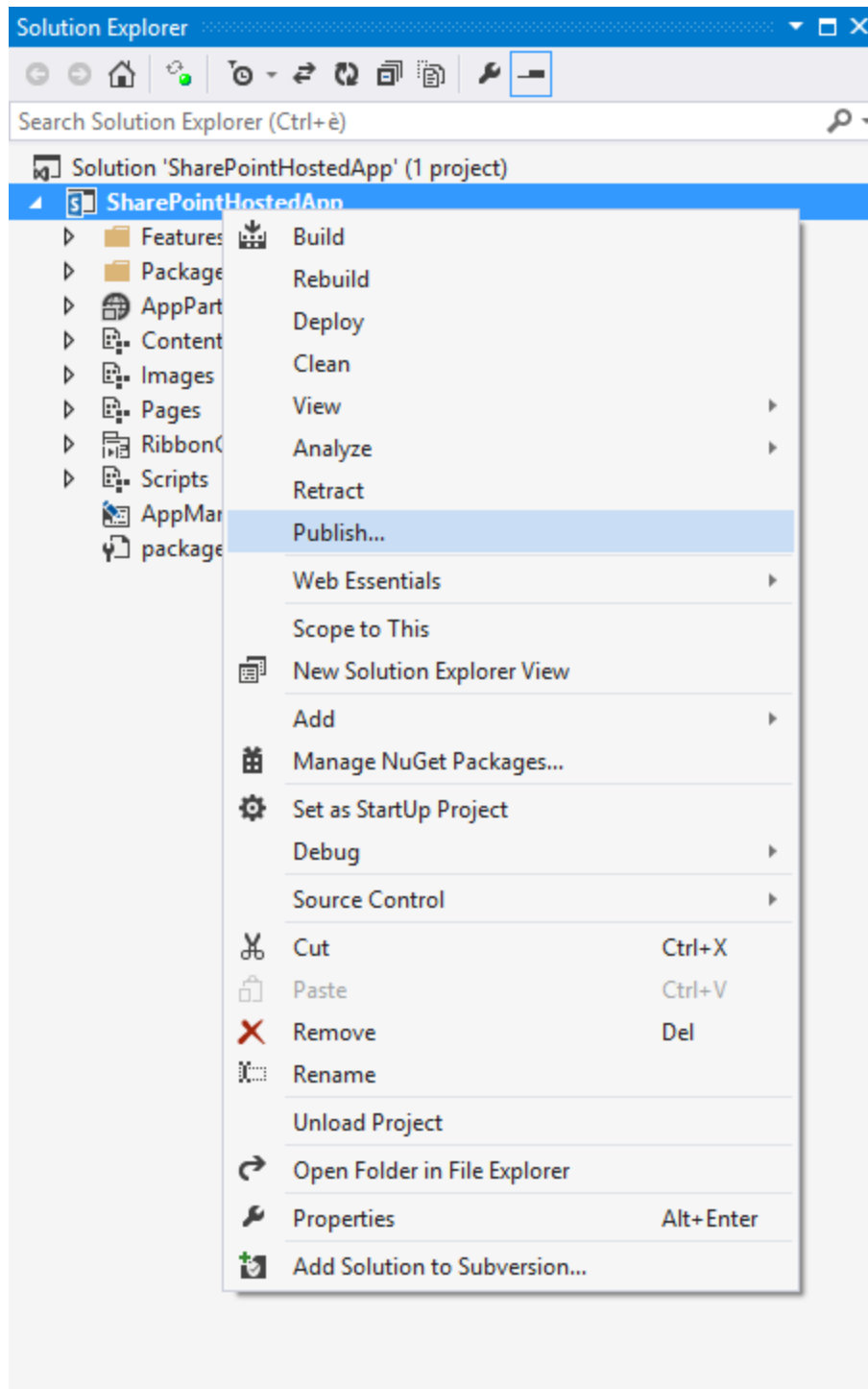


Figure 43: App publishing

Depending on the type of app that we've created, different information will be requested.

SharePoint-hosted

In this case, the publishing process requires that we only build the app package.

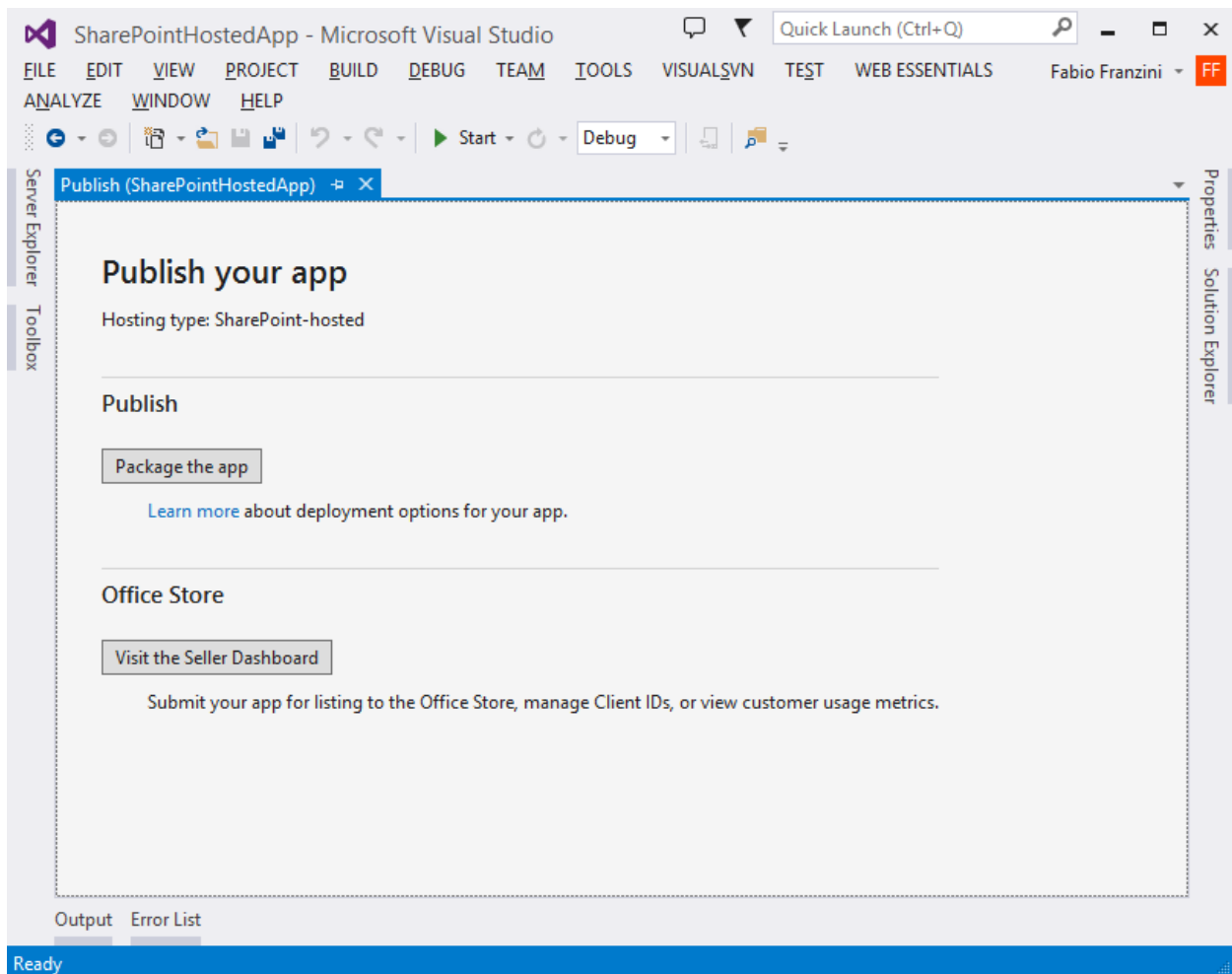


Figure 44: Publish SharePoint-hosted app

Provider-hosted

In this case, the process involves creating a new profile part for publishing, where we will be asked for information concerning the ClientId and client secrets that will be used as the app identifiers in SharePoint:

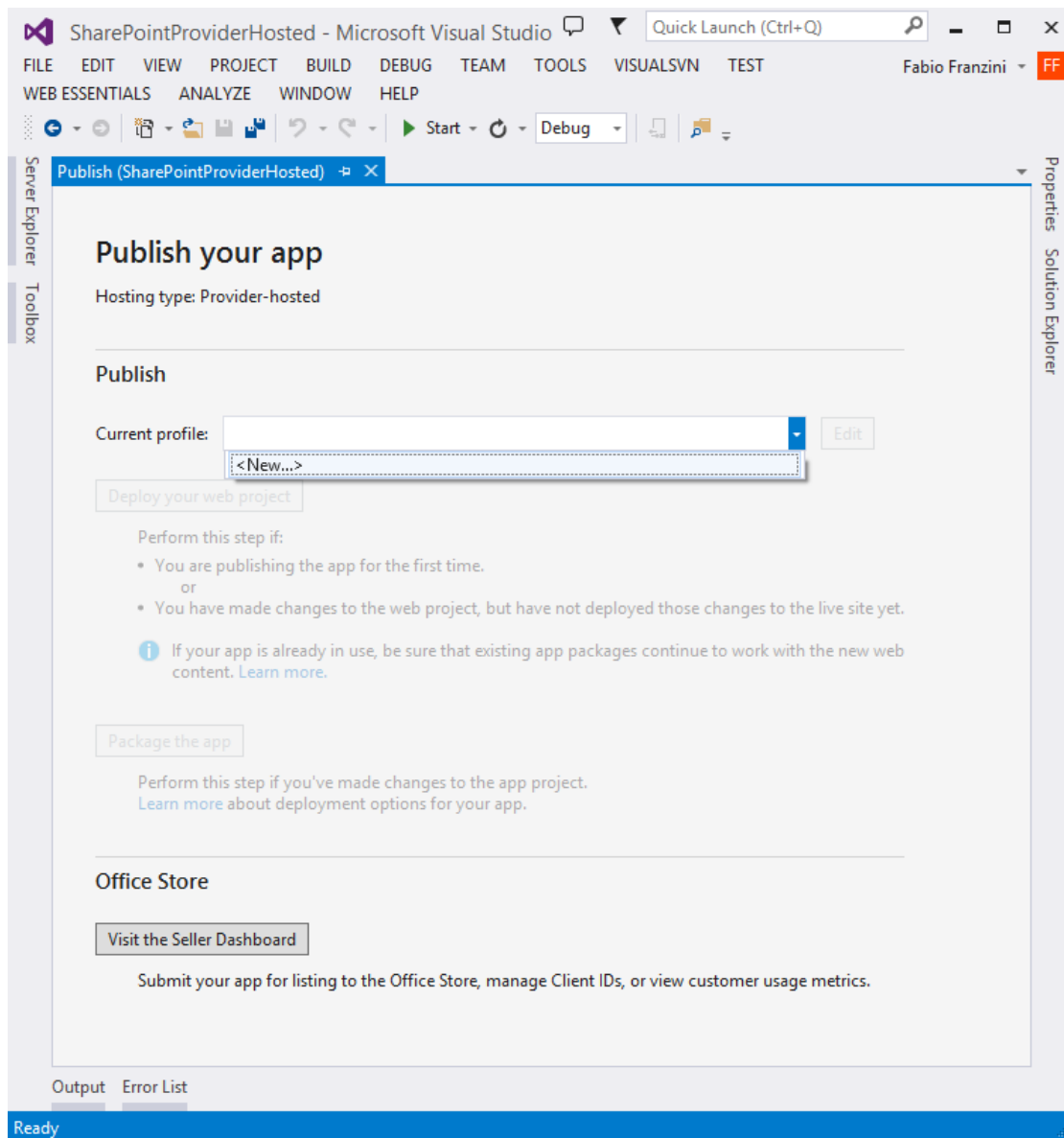


Figure 45: Publish SharePoint-hosted 1

Once we have created the profile, we can both publish to the web and generate the app:

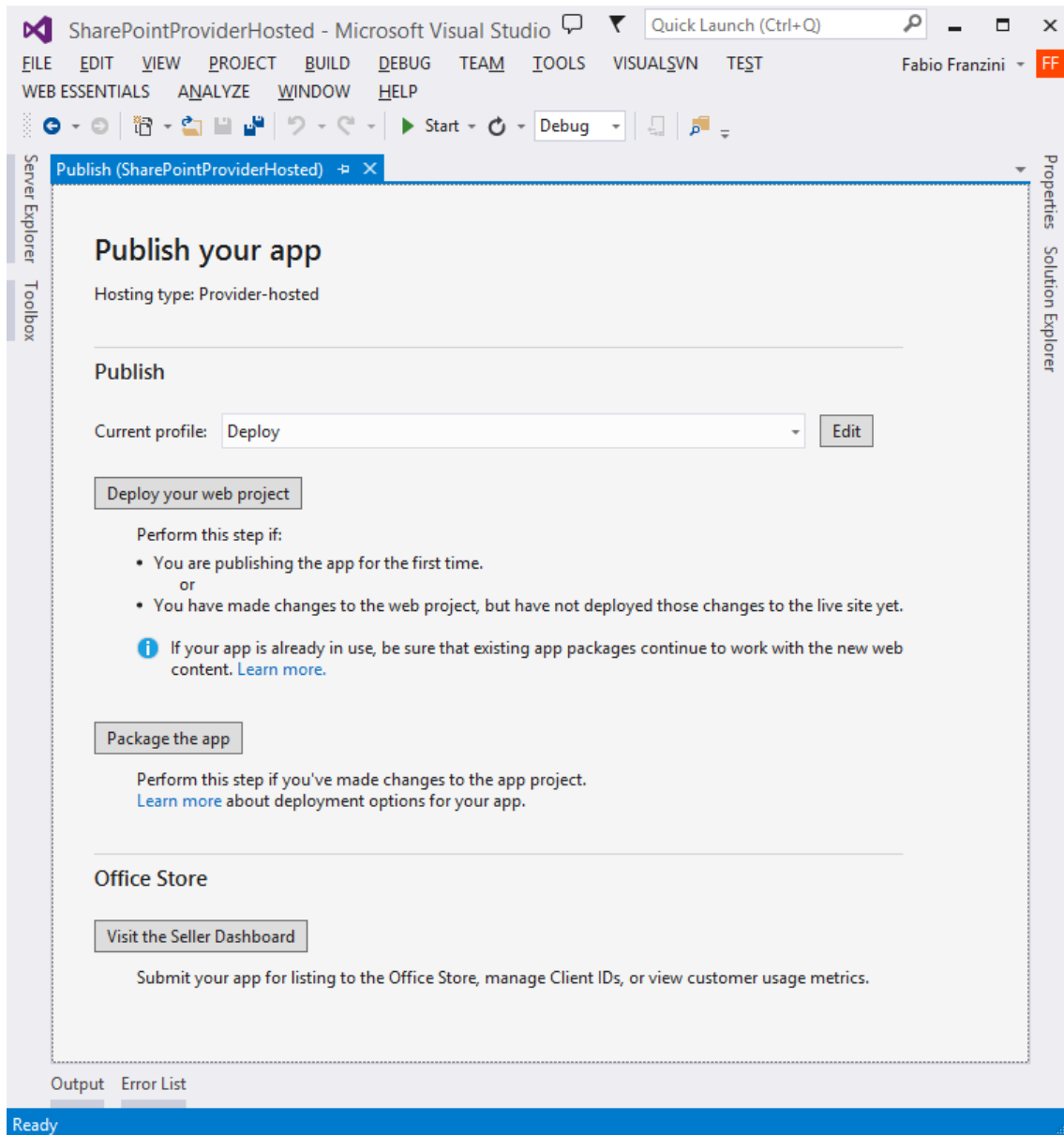


Figure 46: Publish SharePoint-hosted 2

App Catalog

The App Catalog is, as was previously mentioned, one private app store within SharePoint. Technically, it is a site collection that contains two main libraries:

- Apps for SharePoint: SharePoint apps
- Apps for Office: Office apps for 2013

Each library contains a number of columns whose purpose is to describe the app itself so that the infrastructure of the app may allow the installation of same. It shows the user's information, such as title, description, etc.

To create an App Catalog, we can access the Central Administration in an on-premises environment or in the SharePoint administration portal on Office365 as shown in Figure 47.

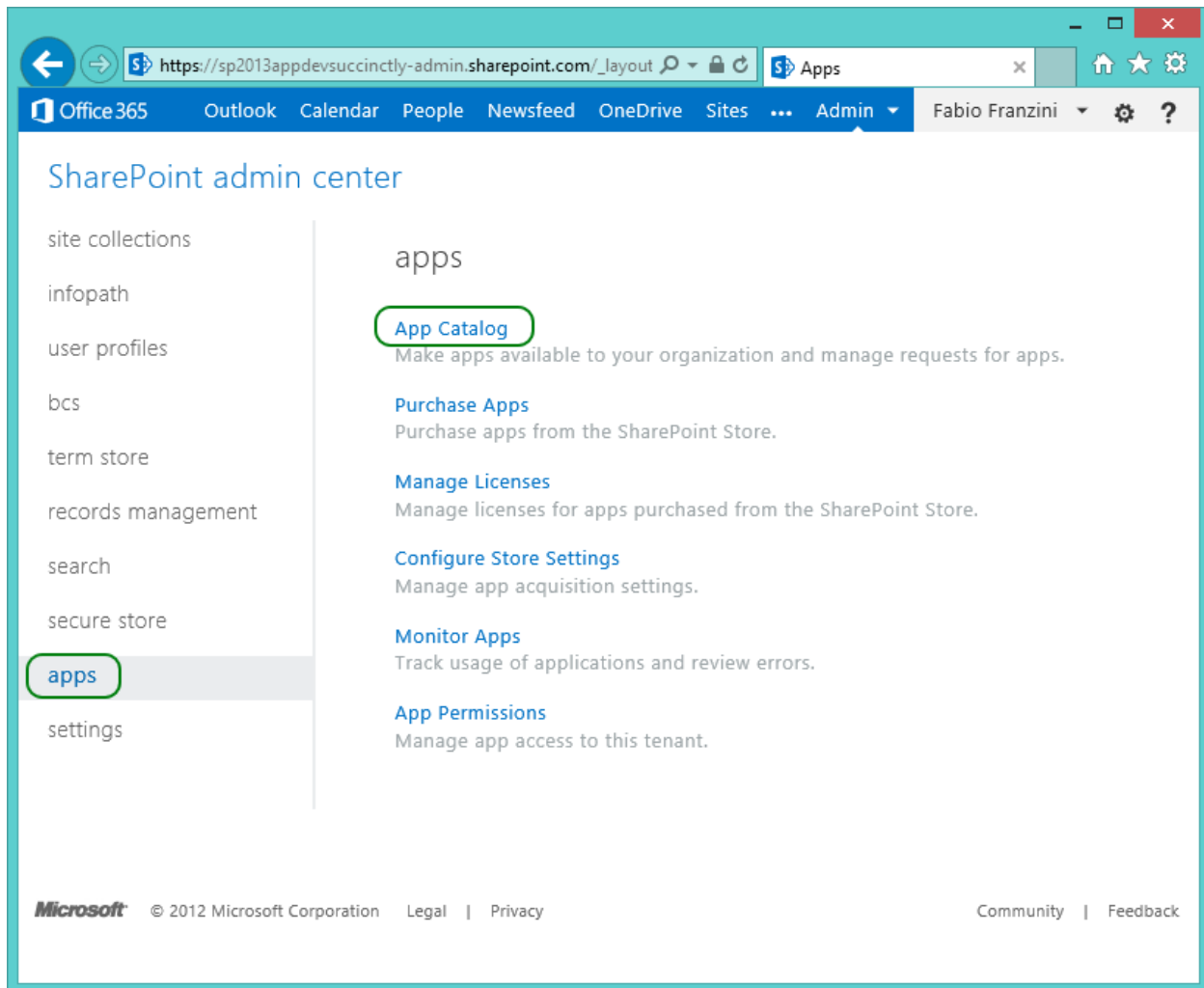


Figure 47: Apps settings

In this case, we create a new App Catalog, and set the settings to create the Site Collection that matches it:

The screenshot shows the 'Create App Catalog Site Collection' page in the SharePoint Admin Center. The browser address bar shows the URL: https://sp2013appdevsuccinctly-admin.sharepoint.com/_layouts/15/online/CreateAppCatalog.aspx. The Office 365 navigation bar is visible at the top with links to Outlook, Calendar, People, Newsfeed, OneDrive, Sites, and Admin. The user 'Fabio Franzini' is logged in. The form contains the following fields:

- Title:** App Catalog
- Web Site Address:** <https://sp2013appdevsuccinctly.sharepoint.com> /sites/ appcatalog
- Language Selection:** Select a language: English
- Time Zone:** (UTC-08:00) Pacific Time (US and Canada)
- Administrator:** Fabio Franzini
- Storage Quota:** 2000 MB of 14644 MB available
- Server Resource Quota:** 300 resources of 800 resources available

At the bottom right, there are 'OK' and 'Cancel' buttons. The footer includes the Microsoft logo, copyright information (© 2012 Microsoft Corporation), and links to Legal, Privacy, Community, and Feedback.

Figure 48: Create new App Catalog settings

Once we have completed this, we also have a new site where we can manage a catalog of all the apps we make available in SharePoint:

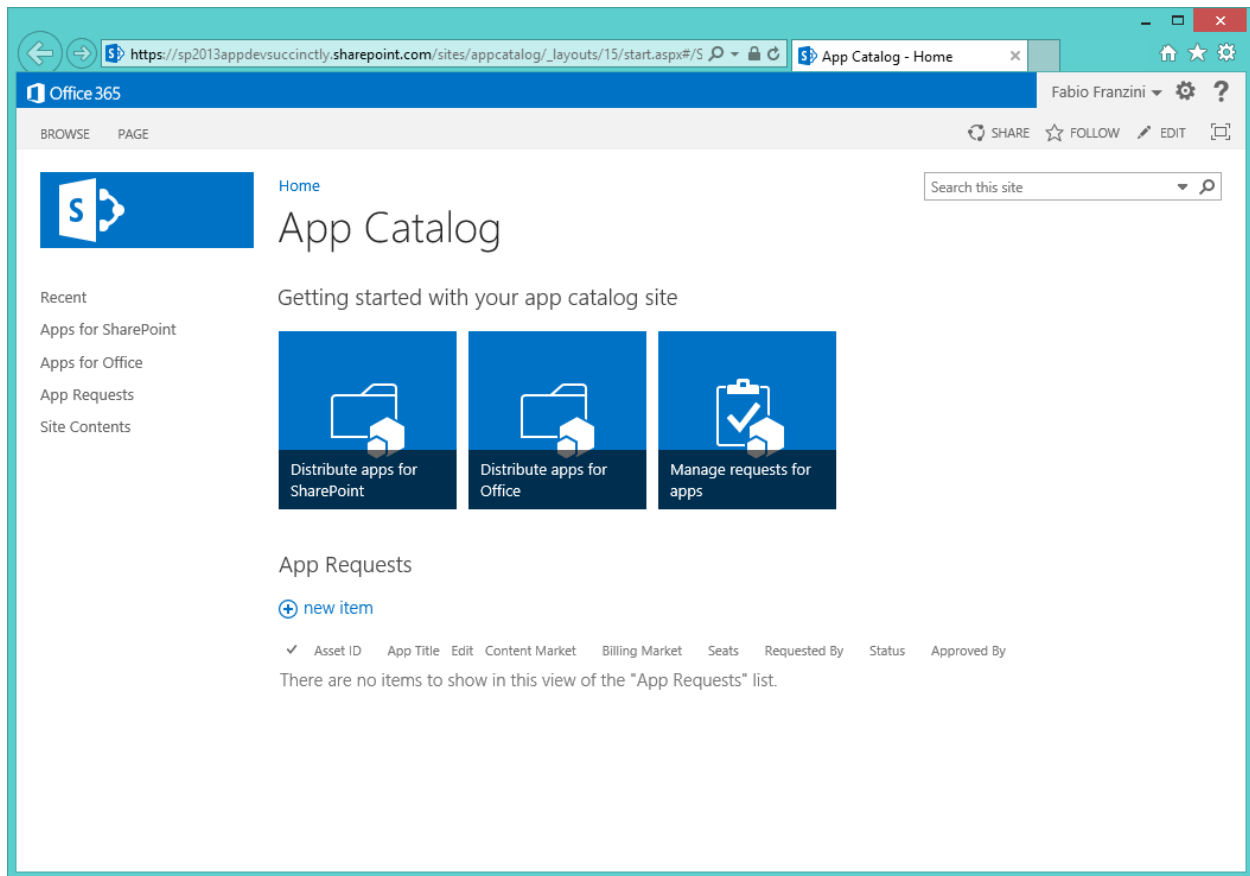


Figure 49: SharePoint App Catalog

By clicking "Distribute apps for SharePoint," we can access the library that will contain the apps:

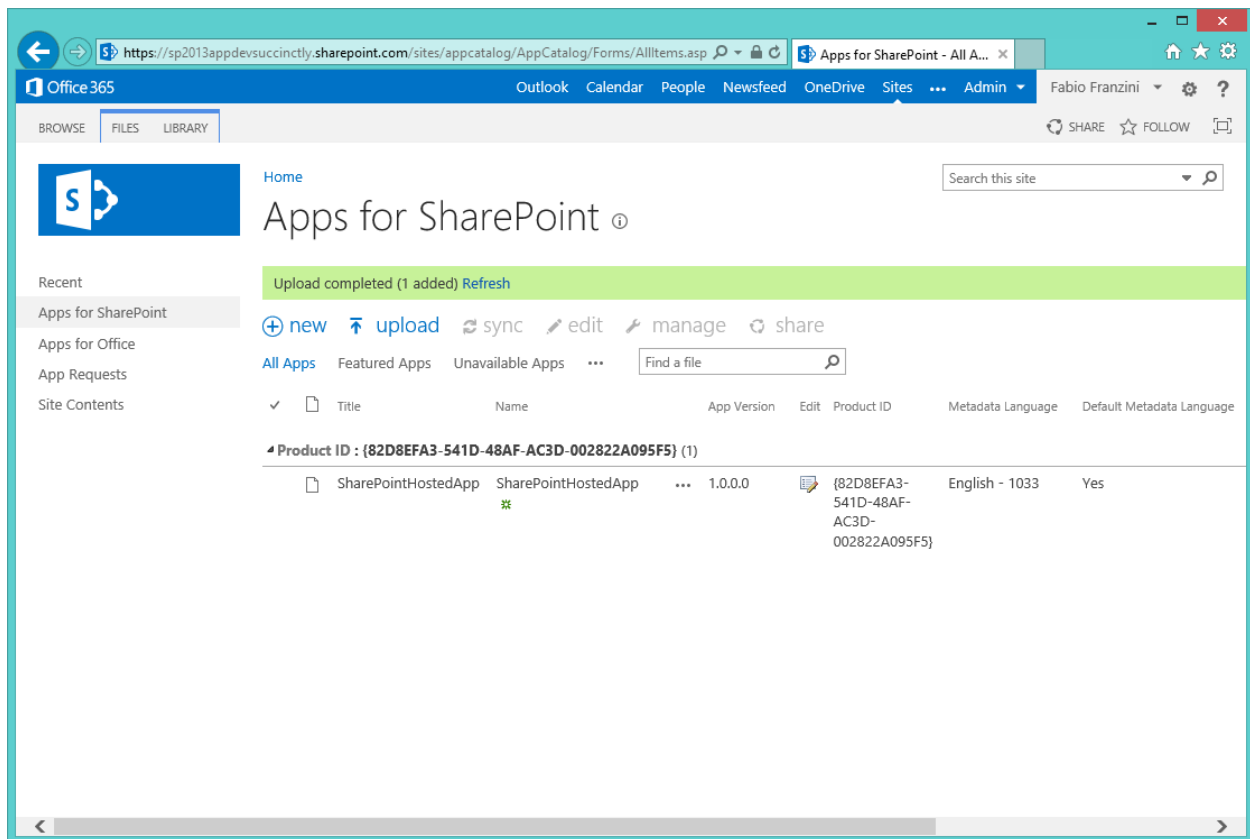


Figure 50: Apps for SharePoint library

Once this is done, the deployed apps will be available for installation by users.

Office Store

By releasing an app in the Office Store, we make our app available to the world.

The first thing to do in order to publish an app is to create an account in the Seller Dashboard. Just go to the Microsoft [Seller Dashboard](#) and enter all the required information.

You can create two types of accounts:

1. Private: For creating an account not tied to a company.
2. Company: To create an account linked to a company.

Once the account is created, you will be able to publish the app.

Before it is made available to the public, the app will be subject to an approval process that validates a set of rules that must be respected and that are identified in the portal itself.

If validation is successful, the app will be released. If not, the address will be delivered to your account with a report that details the validation failure.

Summary

In this chapter, we learned how to publish apps for SharePoint, as well as create a private store that's useful when we want to publish an app within our company or publish on Microsoft's public store.