Connected Services, the design principles behind creating a Modern Service Reference Configuration

Visual Studio .NET (7) initially shipped in 2002 when XML Web Services were the new lingua franca for connecting systems together. A lot has evolved since 2002. Modern services use various technologies, or variances of various technologies and the plethora of devices to be connected continue to expand. Service authorization has changed, such as OAuth keys and tokens requiring configurations of the service for clients to consume them. The days that all services must abide by a Visual Studio service and metadata standard have passed. It’s time for Visual Studio to support various service types and their unique configurations.

Visual Studio 2013 introduced the concept of **Add Connected Service** for Office 365 and Azure Mobile Services APIs. With initial success, and the desire to expand the breadth of services, we took a second look and realized there’s something to this new design. In Visual Studio 2015 we’ve reimagined the Connected Service 1.0 experience, learning from what shipped, with new expectations for new providers and applied the lessons of previous tooling experiences.

Since we wanted to support the breadth of services, how do we validate we didn’t build yet another Microsoft focused tooling scenario? In 2014 we [announced a new partnership with Salesforce](http://www.salesforce.com/company/news-press/press-releases/2014/05/140529.jsp) and thought what a great way to validate Connected Services since Salesforce has been arguably one of the most successful SaaS providers.

In this article we’ll dig deeper into the principals we used when designing Connected Services v2 including:

* Developers should have a common entry, exit and update experience for working with services creating muscle memory and improved discovery for the breadth of new services coming available
* Visual Studio Service consumption should support various technologies (REST, OData, WCF, …) and authentication of the service endpoint, regardless of which standard Microsoft is focusing upon
* Developers should choose from options and Visual Studio should remember their choices, not require developers to repeatedly enter information already known, or copied from their OneNote or saved Email
* Services should be able to express their unique value, ask their unique questions, guiding the developer through the relevant choices as all services are *not* created equal
* Service Providers should have flexibility to consume their metadata in any means that makes sense for their service. As long as the developer can configure the service consumption, it doesn’t matter how the service exposes or finds the required metadata
* The result of consuming a Connected Service should produce the same code developers would have written, over the same runtimes they would otherwise use, rather than implement new runtimes, promising to make all services similar or “easier”
* Code added to the developers project should be scaffolded into the project becoming the developers code rather than the generated into, [“Can’t Touch This”](https://www.youtube.com/watch?v=x8H2-YZUw40) style code
* Developers should be able to customize the scaffolding using T4 templates, or razor templates as relevant in ASP.NET v5 projects
* When something goes wrong, developers should be able to view and fix *their* code just as they would do with any other code asset and, not be left to edit model files (.xml, .edmx, .lsml)
* Sample code should not be added to the project by default, however it should be readily available and relevant to their task
* Upon completion, the developer should be presented a web page providing guidance for: What Happened, optionally any required Next Steps and Getting Started/Sample content
* Developers should be able to reconfigure the service, add new objects, update consumption of existing objects that may have changed
* Service Consumption should follow a “no-turd” design. Artifacts added to the project should be targeted towards the developer editing them directly. Having tooling over code and configuration is an optimization
* Developers should progressively learn and the tooling should be just as valuable the 50th time they’ve used it as it was the first. Once a developer learns how to consume a service, it should continue to be easier and more productive to use Connected Services

We hope developers will find the new Connected Services patterns applicable to their development. And we hope that Connected Service Provider Authors will follow these principals to deliver a consistent, compelling, rewarding experience growing the confidence in Connected Services, the breadth of services available and ultimately the productivity of the Visual Studio developer building service and device apps.

# Why a new Add Service gesture?

If the above doesn’t identify the need to redesign the 12 year old service consumption scenarios, let’s look at the modern services, the technologies used, and the configurations required:

* While many modern services are RESTful interfaces, including the OData services, there are many versions of OData and even variations on different versions of REST. Should all services be implemented with exactly the same pattern? And who decides that pattern? What if someone evolves the “standard”? Should an evolving service be limited by a standard, or what the common tooling currently supports? Does Visual Studio have to chase the evolving standards, and must developers wait for a new version of Visual Studio to consume their new/evolved services?  
  Looking at Dynamics CRM and SAP, who both implemented the OData standard, they both struggled to fit their unique values into the OData standard. SAP, who’s on the OData standards body implemented some of their metadata using custom annotations. Dynamics CRM has so much metadata, developers need to call their SOAP API to discover the robust set of information. Should OData support all these unique scenarios? It would be great if and when it does, but should tooling the service consumption have to wait until the new standard is defined and all the services implement that standard? Or, can Visual Studio support unique configurations and allow the services to evolve at their own pace, not blocking developers along the way?
* What is the metadata standard for REST endpoints? REST has evolved as a modern standard, specifically because of its lightweight nature, applicable to devices where bandwidth is constrained, or needs to be optimized. Metadata formats are evolving, like Swagger, WADL, RAML. But not all services implement them, and if they do, which one? Are they limited by the standard supported by the single Visual Studio tool?
* OAuth requires configuration for each app, including keys and tokens. Modern services no longer use a simple username/password or certificate. Each app has individual configuration on the service, allowing apps to consume services in their own security context. Regardless of the user’s permissions, the ServiceDelivery app may only have read access to the Route objects, while the ServiceAssignment app may have Create and Update access to the Route object. While metadata standards are surfacing, what is the service configuration standard?

We believe it’s time to provide a standard extensibility point for services to plug in a configurator specific to their service and thus, Connected Services v2.

# Digging Into the Design Principals

To provide a bit more context, let’s drill into some of the specifics of our list above and how we envision solving these challenges.

## Balancing consistency over uniqueness. Are all services unicorns?

**The least common denominator approach**

There’s been a historical approach to have a highly consistent design principal that requires all consumers to adhere to strict implementations controlling the experience for reasons from security, consistency, branding and sometimes thought to be ease of use. What we’ve found is this approach can be over limiting, excluding some, while some create strange implementations to work around the constraints, or a constant request of specific extensions that can become over or under engineered. A great example of this is ye’ ole Server Explorer. Server Explorer a classic example of a Visual Studio extension allowing other Servers (services) to plug into Visual Studio. [DDEX](http://msdn.microsoft.com/en-us/library/ms379576(v=vs.80).aspx) was conceived by some brilliant people, some of who are still on our team. The approach was to provide a means for all data providers to express themselves, allowing them to take advantage of the tooling we created for SQL Server. For those like Oracle who invested a lot of time, they got a pretty good implementation. However, Oracle is not SQL Server. They have unique features. That sort of the reason two companies exist for similar scenarios. Companies find unique experiences that differentiate them. Oracle struggled to express their unique features in Server Explorer as they didn’t fit in the SQL Server view we generalized. We spent so much time focusing on building a common denominator that nobody was happy. Providers had to choreograph such a dramatic dance just to participate, and for a least common denominator experience that questions the value. While the experience was built around SQL Server, even the SQL Server tools team bailed on Server Explorer and created [SQL Server Object Explorer](http://msdn.microsoft.com/en-us/library/ms173849.aspx).

**Everyone is a unicorn approach**

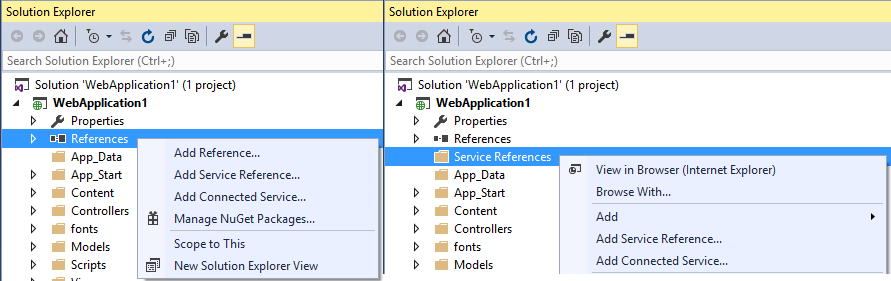
The other approach is each service uses the Visual Studio SDK to create their unique experience. SAP attempted this with their Add SAP Reference to work around the limits of the generic Add Service Reference. Teams like App Insights added context entry points for Add Application Insights with configuration files in the root of the project system. While these solve some problems, they create others. For each experience, the developer must hunt for where the unicorns live. We’ve all seen the Context Menu fill up with everyone’s opinion of why they’re the most important thing to be seen first. It’s all about number of clicks isn’t it?

## Finding a balance

For Connected Services, we felt it important that developers have a common entry point. One developers will become familiar with and one that serves the needs of supporting individual configurations. We identified the places of consistency, and the places of uniqueness. We refer to these as the “black holes” services can use to express their unique configurations.

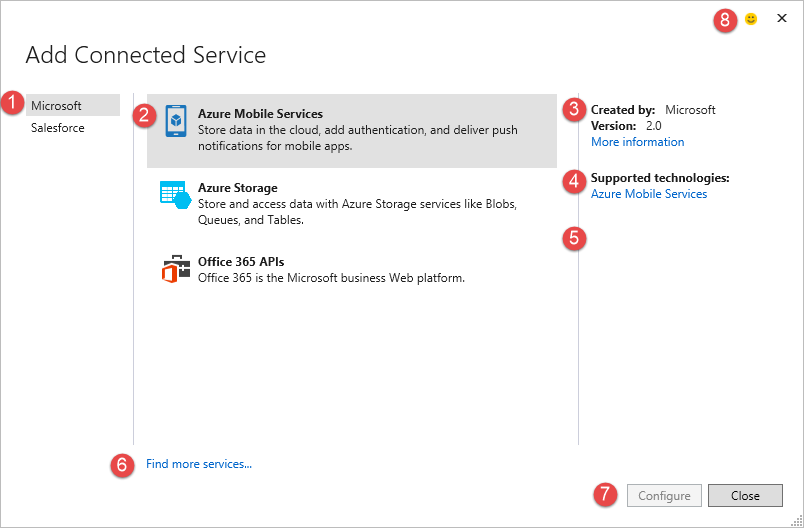
We believe there should be a common discovery experience for developers to find a set of services. We believe there should be an easy way for developers to find and install new service configuration experiences, not limited to the latest Visual Studio Update.

When developers right-click on the References folder they’ll see **Add Connected Service**. This is alongside the legacy **Add Service Reference** as we believe Connected Services are the modern way for Services, so we wanted to use the entry point developers are used to without [moving the cheese](http://amzn.com/0399144463) too far. Starting with Visual Studio 2015 RC, developers will also have the entry point on Service References. Cordova Projects will also have access, however Cordova developers expect the folder to be named **services**.



# Connected Service Provider Selection

Once launched, the developer chooses from a set of Connected Service Providers.



1. **Service Categories**On the left we show a categorized list of services. This is a string based implementation that provider authors can use or honestly abuse. This is an example of where we opted to be more open with this risk of regret. Since our focus centers around enabling developers to find services they wish to consume, we believe they know they’re looking for a Microsoft, Salesforce or SAP Service. We also believe that multiple companies will want to author various providers for Microsoft, Salesforce and SAP.
2. **List of Providers**  
   These are the specific providers for a given category of services.   
   We hope to see companies like rssBus and DataDirect author providers for Salesforce. What we hope to see is rssBus and DataDirect list their providers under Microsoft, Salesforce and SAP. We hope we won’t see rssBus or DataDirect in the left pane because we want developers to think of the service their consuming, not the implementation detail. However, we do know that’s a decision for the provider, so we hope to see the Salesforce providers from rssBus, DataDirect and Salesforce listed under Salesforce for instance
3. **Author and Version Information**   
   Strings to help developers understand who authored the specific provider. The version represents the Connected Service Provider version, not the underlying Service version as we assume services will rev their providers independently from the service. In some cases the configuration may improve over the same version of the service, or the service may version requiring an update to the provider. When you’re asked which version of the Connected Services tooling you’re using, we want this version to be the thing you reference
4. **Supported Technologies and/or Project Types**This turns out to be a very interesting thing for developers to know. What services does this provider actually work over? For the Microsoft APIs above, it’s fairly obvious. But what happens when vendors ship tools like Side Waffle, Goofy Ball. What types of services do these providers support anyway? While we haven’t built any yet, we also see the scenario where an Provider might consume multiple services. For instance someone might write a Salesforce provider that uses Azure Redis Cache to minimize their Salesforce API usage and improve local performance.
5. **Supported Project Types**There’s another aspect that’s missing here, and that’s what Visual Studio Project types does the provider support? This turns out to be a tricky problem. Visual Studio Project Types come in various flavors. The same core project type that’s used for a Class Library is also the same project type used for Windows Forms, WPF and others. This is a gap we’re still working on, so for now, you won’t see what Project Types are supported. But you will notice when something isn’t supported, the configuration button will be disabled. While we’re not sure we can complete this by Visual Studio 2015 RTM, this experience is something we’re thinking of improving.
6. **Find more services…**  
   One of our primary goals of extensibility is expanding the ecosystem, and not couple the support of specific services to a specific Visual Studio release. *Find more services* goes out to the Visual Studio Gallery, which filters to the [Connected Services](https://visualstudiogallery.msdn.microsoft.com/site/search?f%5B0%5D.Type=RootCategory&f%5B0%5D.Value=tools&f%5B0%5D.Text=Tools&f%5B1%5D.Type=SubCategory&f%5B1%5D.Value=tools_services&f%5B1%5D.) category. Which surfaces another gap where we technically don’t limit Gallery Components for which can tag their extension as a Connected Service. At preview, the category was simply titled Services, so who isn’t a “service” these days, and that caused a lot of confusion for developers who didn’t understand why these were listed, and frustration for extensions that posted to this category, and were removed.
7. **Configure - *maybe***  
   Finally, developers can select a provider and configure this bad boy. Well, maybe, maybe not. This is another area we’re experimenting with. Connected Services targets all project types. However, not all project types are created equal. Since Connected Services supports various technologies and configuration, Connected Services doesn’t have a one size fits all configuration approach. This means all providers must implement configuration for all the project types they support. In reality, not all providers will support all project types. So, how does the developer know if the provider works for their project type?   
   We believe developers start with a specific project type, and then look to consume a service. They right-click on Add Connected Service and search for the service they desire. What if Salesforce doesn’t support Cordova projects? Should we show the Salesforce Provider, but disable it? Or, should you keep searching, wondering where it is? We call this the “found the dead end” scenario. When you work with software, you typically interact with menus and navigation of some sort. In many cases, the menu/button is either enabled or disabled if it’s relevant to the selected context. For now, we believe it’s better to show the developer you found what you’re looking for, but it may not work for this particular project type. What we’d like to do is help provider authors know developers are interested in their provider supporting a particular project type with real time analytics, giving them insight. Yeah, that’s the goal. We’re not there yet, but we hope to be. So, for now, we’re simply showing the configure button enabled or disabled if the provider is supported for the active project type.   
   As a side note, tied to #5, this is an area we hope to improve the experience. But we also believe much of the frustration isn’t knowing a specific provider isn’t supported, but rather developers want support for their project type, so it’s only slightly better to know why it’s not supported.
8. **Send a Smile, Send a Frown**  
   In the world of real time telemetry, and being closer to our customers, we have built in the Send A Smile feature directly into the Selection and Configuration of each connected service. When developers submit a smile/frown, the feedback is tagged with Connected Services, so the feature team can easily find the feedback amongst all the Visual Studio feedback. So, please do give us feedback. We do look at it, and do engage customers for more info. Including screenshot with your submission is surprisingly helpful for us to gain context.

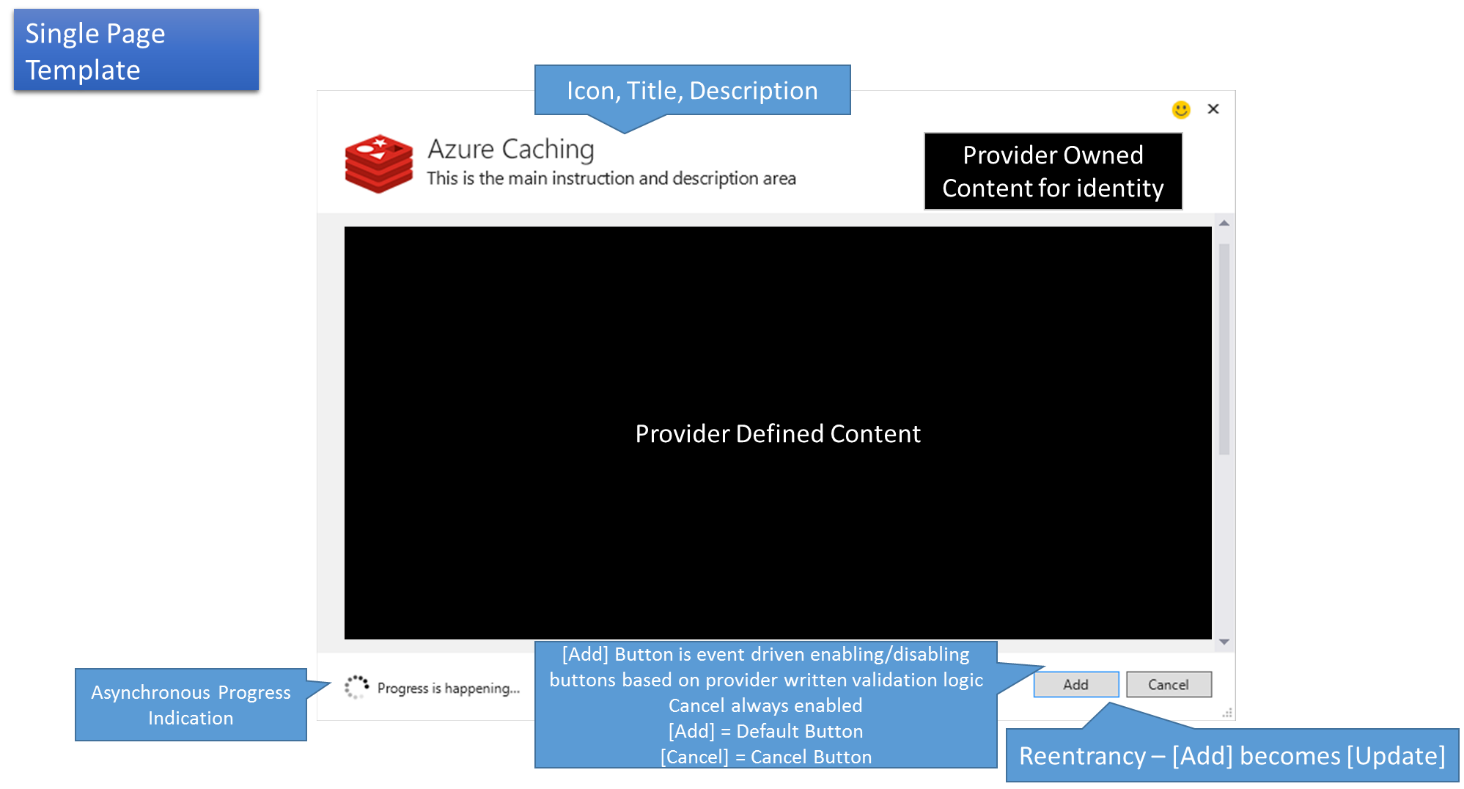
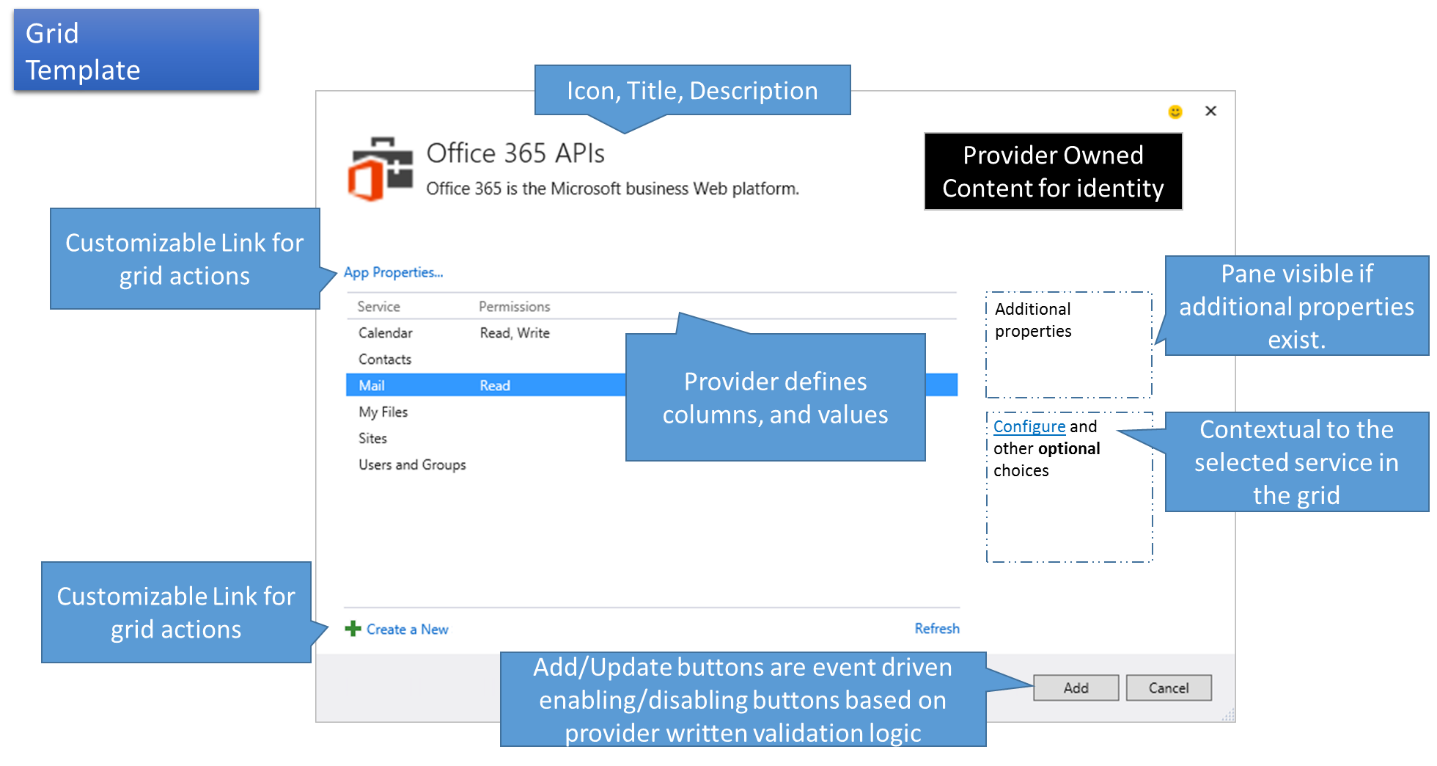
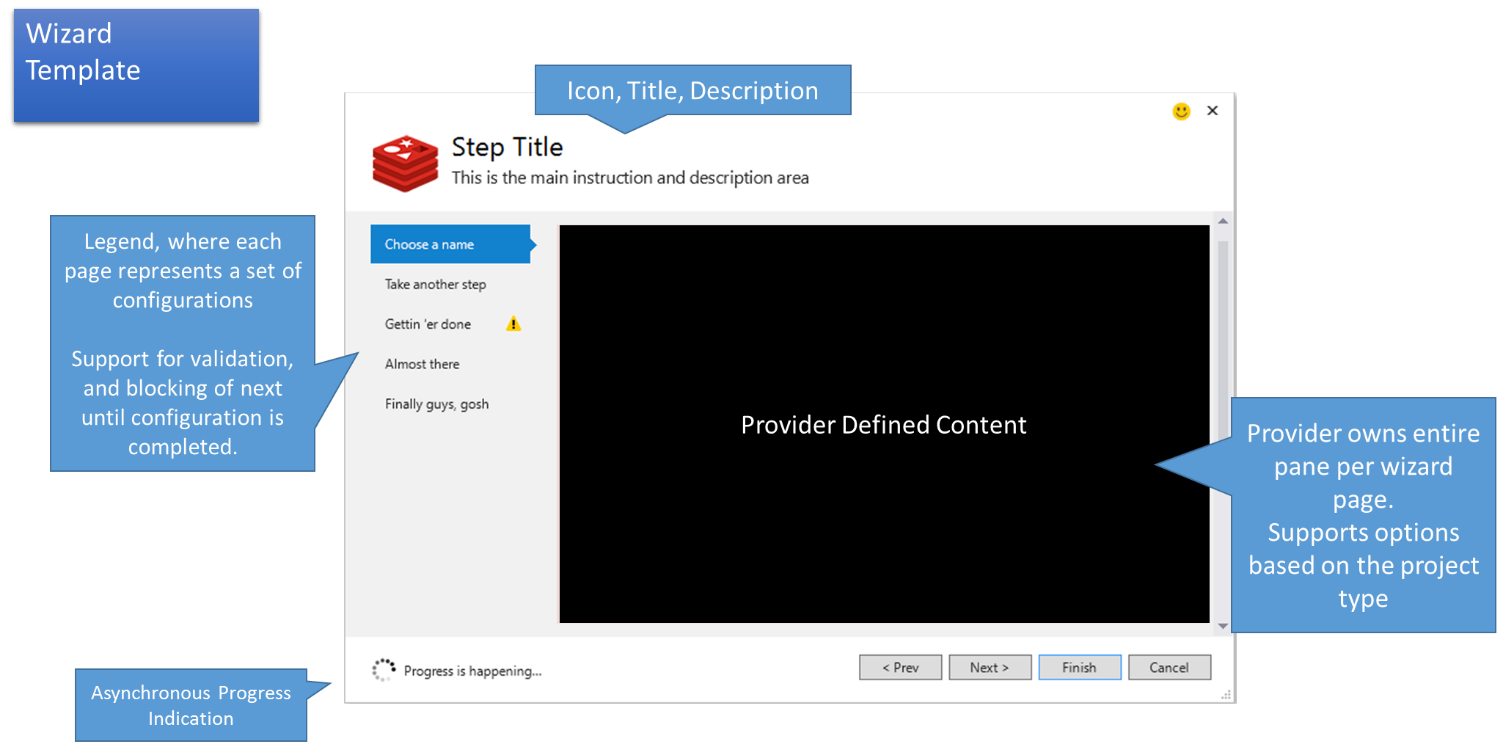
In the above examples of the Service Selection pane, you can see there really isn’t much customization supported, beyond the names, icons and urls for more info. The initial experience is where we believe consistency is most important. But, this is where things start to open up, and providers can venture into the black hole.

# Provider Configuration – Choose Your Weapon

When we looked at the different service configuration scenarios we saw there were simple and complex configurations. In full transparency, we heavily debated this and were trying to find a single configuration experience that “will” fit all services. This was one of the constraints of the v1 experience for Connected Services. For those that use Azure Mobile Services, the grid experience was fairly natural. For those that tried the Office 365 experience, you likely found it a hunt, peck and trial for how you should interact with the grid as the Office 365 team was “forced” to fit within the grid experience.

This is where Salesforce was the most helpful. There were several steps. Steps that didn’t adhere to a common authentication model of Azure. There were multiple environments to connect to, multiple authentication strategies to use and the selection of which objects developers wanted to opt into for their project consumption.

To provide the breadth of simplicity and consistency, with the flexibility of more advanced services, we settled on 3 templates:

1. **Single Page  
   **  
   The Single Page Template provides a common heading, progress bar and add/update and cancel buttons. The “black hole” in the center is completely owned by the provider. This is their sandbox to play in.
2. **Grid**  
   ****  
   The Grid template has a lot more structure. Providers can extend the columns and provide various elements, but they don’t get to paint much of the UI. For many, this is the easier implementation as providers just plugged in their information. If providers have a list of “things” the developer would select from, and occasionally create a new thing, and there is much of anything else to configure, this is likely the template for you.   
   Many of the Azure services use this template and it fits them well. For Visual Studio 2015 Preview, Office 365 chose to keep with this grid, even though we knew it didn’t quite fit it based on our Visual Studio 2013 and Connected Services v1 feedback. So, we expect them and a few other providers move to one of the single page or wizard templates as they wanted more flexibility to present their options.
3. **Wizard  
   **  
   The Wizard is what you’d expect. A multi-step version of the Single Page template. There are a set of elements that are customizable in various levels of detail. There’s a legend of breadcrumbs enabling skipped navigation with feedback of configuration errors. We hope developers will find the perimeter (chrome) consistency helpful, and the providers will have the “black hole” for the flexibility they require to provide a consistent experience. We did choose not to include the identity black box in the wizard as we found it weird to support changing identity across a set of configuration pages.

## Common Elements of the templates

Each template has an Icon for their logo, Title and Description to explain that particular page their on. We also surface the Send A Smile, and similar to the Connected Service Selection UI, when developers send a smile/ frown, we also capture the name of the provider.

# Principles for Provider Authors

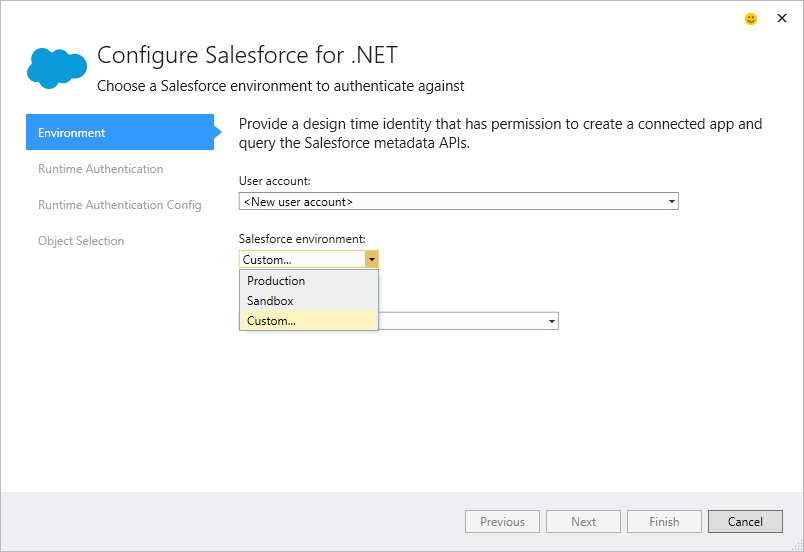
With the big black hole, and full access to the Visual Studio Project System, provider authors can do just about anything to your project. We joked that we’ve enabled developers to create Candy Crush within Visual Studio. However, that’s not really the goal. The goal of Connected Services is to provide a consistent feeling set of experiences. Services are writing Providers because they want to fit in the Connected Services experience. Rather than implement hard contracts for each provider to abide to, we’ve taken a much more open approach, which means we need to outline what are the specific principles. We believe providers want the flexibility, want to adhere to the guidelines for consistency, but need to know what they are.

# Discovery and Selection based Configuration

A core concept of Connected Services aims to help the developer with information already known. The developer chooses from options, rather than enter information they already know. If they must enter a value, the values are remembered. As the previous consistency and template sections outlined, we expect developers will know the service type they’re consuming, and won’t need to know the URL of the service. What we want is for developers to choose, not remember/copy/paste/type.

# Selection based configuration

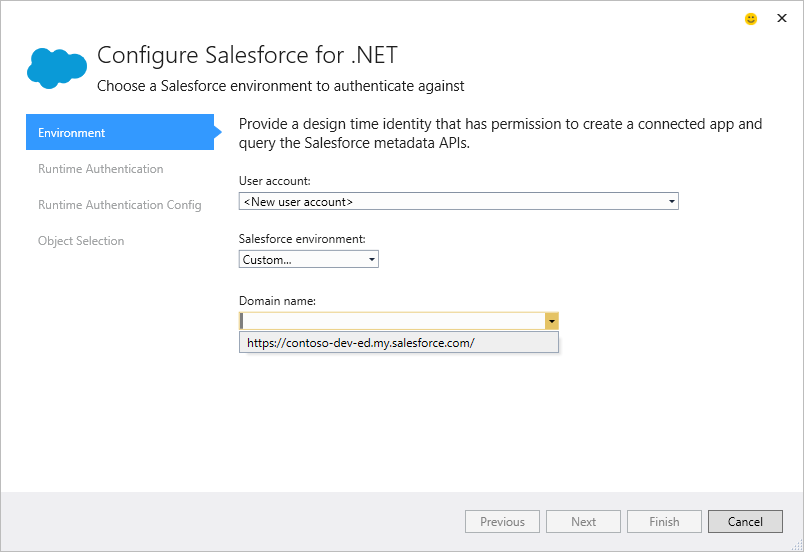
Modern service almost always have some sort of discovery mechanism for discovering the their services available. Rather than entering a URI of some sort into that a common textbox, the developer provides their credentials to the specific service. Using Salesforce again, we provide our developer credentials. By selecting the environment (Production or Sandbox), the Salesforce Connected Service provider knows where to authenticate the developer. If the developer is using a custom endpoint, they can enter that URL. However, they only have to enter it once as the Salesforce Connected Service provider will remember the URL for subsequent configurations, leveraging their investment.



# Great Memories

When the developer does enter specific values, the provider should remember those values for subsequent configuration. In the case of Salesforce, the provider remembers the developer has previously used the <https://contoso-dev-ed.my.salesforce.com/> custom endpoint and allows them to choose from the remembered URLs.

These MRU values should also roam with their Visual Studio profile. Referencing the book by Nir Eyal [Hooked: How to Build Habit-Forming Products](http://amzn.com/1591847788), as the developer invests their time with Visual Studio we want them to be rewarded as they use their Visual Studio Identity on other machines or VMs.



# Config and Coding Patterns

This is where we expect developers will reap the biggest reward from using the modern Connected Services experience. Add Service Reference used a [Single File Generator](http://msdn.microsoft.com/en-us/library/bb166817.aspx) that took the meta-data from the source service and generated a chunk of proxy code into your project. As the name implies, it generated a single file. Also as the name implies, it’s generated. And can be regenerated. But also subtle in the name and the implementation was the file wasn’t quite “owned” by the developer. Developers had the responsibility of the file, without the ability to edit it. Service files were hidden files, placed in the developer’s project. The developer must compile the file, check them into source code control. But, if they wanted to edit the contents of the file, they didn’t have any practical solutions. Sure they could edit the hidden file, and as long as they didn’t re-run the service, it was ok. Sort of, as the code in the file wasn’t really intended to be developer friendly, it was designed to support the extensibility of the file through partial classes, partial methods, and events raised for things like XXXPropertyChanged. As a result the files were humongous and remembering the great MC Hammer, I like to refer to them as [“Can’t Touch This”](https://www.youtube.com/watch?v=x8H2-YZUw40) style code.

The Connected Service code approach is as follows:

1. Prompt the developer with the questions needed to scaffold the appropriate code
2. Scaffold the code into the project, giving the developer full ownership of the file
3. Author the best practice code the developer would have written themselves
4. Use templates to drive the scaffolding, allowing developers to customize the best practice code to fit their individual or corporate standards
5. Reference SDKs for fixed APIs, scaffold code for customized object models

## Scaffolding

ASP.net introduced code scaffolding, which essentially is an improved Item Template feature where the developer owns the file once it’s been added to the project. Connected Services adopts this model as we’re focused on helping developers write *their* code.

Rather than generate all the code for all the scenarios with Single File Generators, Connected Services scaffolds just the code you need, and since it’s scaffolded into your project, you can evolve the code as you see fit, when you see fit.

Let’s take a simple example of some property definitions:

**Property Definition for Service References Generated Code**

/// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=1)]

public string Id {

get {

return this.idField;

}

set {

this.idField = value;

this.RaisePropertyChanged("Id");

}

}

/// <remarks/>

[System.CodeDom.Compiler.GeneratedCodeAttribute("System.Xml", "4.0.30319.34230")]

[System.SerializableAttribute()]

[System.Diagnostics.DebuggerStepThroughAttribute()]

[System.ComponentModel.DesignerCategoryAttribute("code")]

[System.Xml.Serialization.XmlTypeAttribute(Namespace="urn:sobject.enterprise.soap.sforce.com")]

public partial class Contract : sObject {

private string firstNameField;

/// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=32)]

public string FirstName {

get {

return this.firstNameField;

}

set {

this.firstNameField = value;

this.RaisePropertyChanged("FirstName");

}

}

private string emailField;

/// <remarks/>

[System.Xml.Serialization.XmlElementAttribute(IsNullable=true, Order=23)]

public string Email {

get {

return this.emailField;

}

set {

this.emailField = value;

this.RaisePropertyChanged("Email");

}

}

In the above examples, the Add Service Reference code must generate several attributes, the private backing field, and raise events, even when in most cases there’s no logic to be written. What I didn’t include in the Service Reference, was the additional 217 lines of code for the definition of the sObject.

**Property Definition for Connected Services Scaffolded code**

[Key]

[Display(Name = "Contact ID")]

[Createable(false), Updateable(false)]

public string Id { get; set; }

[StringLength(40)]

public string FirstName { get; set; }

[EmailAddress]

public string Email { get; set; }

The equivalent Connected Service scaffolding is much more compact. It only adds the code necessary to get started. If/when logic is needed within a property getter/setter, the developer can add that code. In this case, the Connected Service scaffolding includes attributes for labels (Display), Creatable, Updateable, MaxLength, Email, URL, Phone attributes. If you were doing a code review for a developer that just checked in some changes, which code would you expect to see?

## Best Practice Code

So this may seem like an obvious goal. Who would place code into the developer’s project that doesn’t follow best practices? Well, it depends on which best practice, and did the tool have enough context to help the developer with what they really wanted to do? Or, was the tool limited without the context, and could only provide a trivial sample? Likewise, if the code was meant to be the output of a Single File Generator, was the intent for the developer to edit that file, or extend it through partial classes, partial methods and/or events?

Because Connected Services provides service specific configuration, and it now scaffolds code rather than generates it, provider authors can now ask the appropriate questions to apply the best practice to each specific scenario and scaffold it into the project.

## Template Driven Scaffolding

The last component is to leverage templates. While a specific service may have the baseline for the best practice code, companies have standards by which they follow. They may implement a specific exception handling and/or logging patterns. They may implement their own variation on INotifiyPropertyChanged. By using customized UI to ask the right questions, scaffolding so the developer owns the code, and templates to customize company or individual preferences, developers can now use Connected Services to optimize their service consumption experience that matches the services and companies goals and standards.

## SDKs and Scaffolding

When consuming services, there’s a balance of fixed and custom APIs the service may expose. For instance, the Office APIs are fixed enabling the provider to add an SDK library to the project, and minimal code to call the various services. There aren’t customizations that need to be accounted for in the API.

Then there’s the Dynamics CRM, SAP and Salesforce APIs. Each tenant customizes their definition of a Customer, Lead, Contact or adds other custom objects. As a result, the project would scaffold out code to interact with those customizations.

There’s a balance to be had for consuming service specific SDKs and scaffolding out customer code. For Salesforce, there’s a base .NET SDK for interacting with the Salesforce APIs then all the Salesforce Objects are customizable so we need scaffolding to reference those objects.

Let’s review a snippet for interacting with Salesforce:

ForceClient client = new SalesforceService().GetWebServerForceClient();

QueryResult<Contact> contacts =

await client.QueryAsync<Contact>("SELECT Id, FirstName, LastName From Contact");

The **ForceClient** comes from the Salesforce SDK. There’s no code in the project that defined a ForceClient. The SDK is included in the project using NuGet, sort of. More on that later.

The **SalesforceService** is a scaffolded/configured type that reads configuration information to return a configured service endpoint. The type was scaffolded based on the authentication strategy the developer chose in the Salesforce Connected Service Configuration wizard.

Lastly the **Contact** type is a [POCO](http://en.wikipedia.org/wiki/Plain_Old_CLR_Object) type uniquely configured based on the tenant’s definition of a Contact.

The design principles of Connected Services enables Provider Authors to use multiple patterns, providing the best practices a developer would have followed if they were to consume the services directly.

# Metadata Driven Configuration

With Connected Services, Visual Studio no longer asks developers to enter URLs into a common textbox. It no longer need to guess the technology used (.asmx, .WCF, OData or REST). Visual Studio no longer needs services to conform to an inevitably dated standard. Instead, each Connected Service Provider is service specific, each provider can find the metadata specific to its service.

Let’s use Salesforce as an example. Unlike Azure and O365 services, which standardize on OData (with $metadata), Salesforce uses a more native REST protocol. However, Salesforce does have a way to retrieve metadata. They call it their [Describe API](http://www.salesforce.com/us/developer/docs/api/index_Left.htm#CSHID=sforce_api_calls_describealltabs.htm|StartTopic=Content%2Fsforce_api_calls_describealltabs.htm|SkinName=webhelp), which is actually a SOAP API. A SOAP API that describes their REST API.

Since the metata information is needed at design-time to construct the scaffolded code, does it really matter how or where the information is retrieved at design time?

Likewise configuring the OAuth endpoint in Salesforce isn’t done with Azure AD (AAD). It’s done on the Salesforce Portal. The Describe API has APIs for creating and maintaining OAuth endpoints, which Salesforce calls Connected Apps. Rather than trying to invent a new standard for OAuth configurations, trying to get the top services to adopt it, then waiting, Connected Services enables Salesforce to build a Salesforce Provider, with the APIs they support today. And, does any of this design time configuration impact how your mobile app connects to the service at runtime?

Let’s say Contoso adds a Region field to the Salesforce Customer object. When Connected Services consumes the Customer object, and finds the Region field, how does it know to generate a property of type String with a MaxLength of 30, it’s required, and the default is “East”?

Because Connected Services is provider specific configuration, the provider can fetch the metadata wherever and however they determine. If the provider knows it’s using $metada, such as SAP, great. If the provider uses a SOAP service like Dynamics CRM, no problem. Each provider author will simply implement the code required to get the metadata from wherever they desire. The appropriate metadata would be exposed to the T4 template used to scaffold the code into the developer’s project, so the developer may customize the code, reading the metadata as surfaced in their template model.

# NuGet

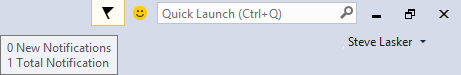
Developers are quickly migrating the consuming of SDKs through NuGet which provides a stable, well known version of binaries and in some cases project assets as well. With Connected Services, the recommendation for consuming SDKs is through NuGet, however there’s a question of stability, performance while enabling the latest version. One question is whether the Connected Services Provider should reference the latest NuGet SDK for its associated service, or should it bind to a specific version? NuGet allows developers to request a specific version, in a known state. This gives developers who publish their SDK to NuGet the ability to make breaking changes when deemed necessary.

The primary goal of tooling is to provide productivity. However, if developers received inconsistent results, including broken experiences when using a productivity tool, would they continue using that tool? If a Connected Service Provider was designed against a specific version of a NuGet SDK, shouldn’t it always default to that version? When the developer finishes their configuration, they’re in a known stable state. Because NuGet is updateable, the developer can still right click and check for updates. WE believe this pattern provides the best balance of developer expectations.

Because a specific provider is bound to a specific version of a NuGet, the next question is whether the provider should download that NuGet from NuGet.org each time. We’ve found this can add a number of seconds when configuring a service, without any benefit. As a result, our design principles are to cache the NuGet on the developer’s machine as part of the Provider.vsix installation.

# Updating Providers

Of course, when there’s a significant update to the service or provider, including the configuration UI, Scaffolding, Templates or referencing a newer NuGet SDK, providers can simply update their VSIX and publish to the gallery. Visual Studio will then prompt the developer for updates in the top right corner.



# Samples – how do I use this thing?

Samples are one of those things that many developers want to get started, but once they know how to use the service, the sample can quickly get in the way. As a Visual Studio developer, what’s the first thing you do when you create a Class library project? Delete Class1.cs. While not quite a sample, if the connected service added sample code to your project after the 50th time you’ve used it, wouldn’t that would be kinda annoying? This is what Office 365 did in Visual Studio 2013, and while it demo’d really well, the feedback was loud and clear, no demo fodder.

While we discussed options for providers having an option to scaffold sample code to the project, we’ve found that providing sample content upon completion of configuration a best case balance.

Upon completing the Salesforce Provider a set of guidance docs are launched. They’re launched in the users default browser, and they can keep them open while working with their project. Providers such as Salesforce implement a copy button to copy the code to the clipboard and paste into the developers project. While not as productive the first time as simply adding the code to the project, developers typically only need sample code the first few times. If they would use the code most of the time, it’s really not sample code, but scaffolding code. Connected Service providers should optimize on the “I’ve learned it, make it productive scenario”, while enabling the getting started developer as well through documentation.

# Delivering Continual Value

One of the early debates we had while designing Connected Services v2 was whether it was simply a primer to your project, or could it deliver value over the life of a project.

Consider the scenario where your company is building a new customer service app. Once past the prototype phase, a set of developers will create a new project, check it into source code control. Someone will define a set of services the app will consume. Development will continue over the next year or two, with hopeful iterations being shipped quite often. Was Connected Services intended to be valuable for the initial consumption? Are the services the app consumed static? What happens if the project consumed Salesforce and a new field was added to one of the objects? What if a new object was defined, an attribute for Email, Phone, URL or a MaxLength was set? Since the project was scaffolded with code to define the POCOs representing those objects, how does the developer update their project to reflect these changes to the service?

This is an area we’re still working on, so I’ll outline some of the challenges and our goals and hopefully you’ll see this implemented in some of our providers soon.

## Generation vs. Scaffolding

When Visual Studio used Single File Generators, it was fairly easy for the tooling scenario as you could simply right-click and choose Update Service to regenerate the “can’t touch this code” with the updated client proxy. This was easy for the tooling, but the developer felt the burden as any customizations to the generated code were overridden. Any code that extended the generated code would possibly be broken if it referenced code that was no longer compatible. For instance changing from an int to a string, or renaming a property or object.

With Scaffolding, the developer owns the code, so we can’t simply replace the code as one of the tooling commandments are: “Thou shall not replace developer code”. So, what to do? It depends. Let’s break the various scenarios down.

**Consume a new object, not already in the project**

In this case, the developer re-launches the Connected Service Provider by selecting Update Service. *Note, as of Visual Studio 2015 Preview, this is not yet implemented.*Using the Salesforce example, they walk through the wizard and are presented with the list of objects. The Salesforce Provider queries the service and has a list of objects current on the service. At the same time, the provider can query the project for the types already added to the project. It sees the Customer and Contact types are already in the project. Of the list of objects found on the service, the wizard checks the checkboxes for these two name matched types. The developer then checks off the Lead object and presses finish. Since the delta is the addition of the Lead object, the provider simply uses the T4 template to scaffold in the Lead type to the Models directory.

**Update a model with new properties**

A new field named Region was added to the Customer object. This time when the developer goes through the wizard, and they’re on the Object selection page, there’s a glyph next to the Customer object indicating something below has changed. Upon expansion, the Region property is highlighted as new. When the developer hits finish, the provider simply evaluates the properties on the existing Customer type and adds the Region property to the existing code. Using [Roslyn](https://roslyn.codeplex.com/) the provider can understand the code in the file, add the property without impacting any of the other code. Will the provider place the new property definition in the exact place the developer wanted? Maybe, maybe not. But what’s easier, moving some code within a file, or having to manually understand the property name, type and any attributes such as MaxLength or other DataAnnotations and having to hand author 5, 10 or 50 new properties across several objects?

**Update a model with changed property attributes**

Using the same Region field above, let’s say someone updated the service to require the Region field, set a max length and even provide a default. When reflecting over code, there’s a delineation between the body of a property, and the definition of the property. In this case, the provider can refactor the property type from int to string without impacting any code within the Setter or Getter. The provider can add attributes for MaxLength, Email, Phone or URL. There’s no guarantee the code that references this type will continue to compile if it were changed from int to string for instance. However if the developer changed these attributes themselves, they’d have the same problem, so this meets our goal of “*do unto the project as one would do themselves.*”

**Deleting properties or objects on the service**

This starts to get a little more interesting, and we still need to spend more time here, but let’s outline this problem a bit more. As it turns out, the Region property was a bad idea to be added to the Customer object. The property was removed from the service. The developer now runs through the wizard. Are they prompted the Customer Type has a definition for the Region property that doesn’t exist on the service? What should be done when the developer intentionally added a property to the type that never existed on the service? What about a computed property for FullName? Should the unmatched properties have a glyph? We haven’t landed on a design principle here, yet, so stay tuned or provide your thoughts.

# No Turds

What is a turd you ask? Must you ask? In the technical definition, a turd is a file in the developer’s project that the developer wouldn’t interact with directly and has no perceived value. They are perceived as having no value, or “stink” polluting the developer’s project and SCC structure. For instance, .config files can be edited by the developer. .xaml files can be manually edited as well, or represented with by a designer. .csproj files are used by the project system and may be edited for SCC merging. These are well understood files that have value and support direct or indirect interaction. Turd files are files that simply fill a gap that Visual Studio needed some additional information, but has no direct developer interaction and quite likely, we didn’t take the time to find a better solution or better bucket to place these in. The csproj.user files are a good example as Visual Studio needed some place to store user customizations that aren’t checked into SCC.

One could argue that turds are in the eye of the beholder. Bill Murray had his own classic perspective in [Caddy Shack](http://youtu.be/TPxiXGr9nFM). We like to think, if it looks and smells like a turd, it’s a turd.



For Connected Services, we worked hard to avoid placing any unnecessary files in the developer’s project. For Visual Studio 2015 Preview, we added GettingStarted.html files to the project, enabling access to the docs if the developer closed the browser page launched upon configuration completion. While there’s some perceived value in the scenario, the file doesn’t belong in the project system as it doesn’t relate to the building of their app. It could be a virtual node, giving the developer the experience of reopening the guidance docs without the “turd” placed in their project. We plan to move that information into the .csproj file to avoid this perceived turd.

# Config over Code

Rather than save off the configuration chosen the first time the developer ran through the configuration wizard for reentrancy, we read the project artifacts. Connected Services follows the design principle that designers are simply helpers for what you’d do if you wrote the code yourself.

For instance, we have a design for updating the consumed objects form Salesforce where we read the types scaffolded into the models folder. Does it matter if Connected Services scaffolded the Customer type, or if someone hand authored it? Once added to the project, it’s the developer’s code, and we have a design principle to work over the developer’s code, and not introduce new model files, or turd files.

Suggestions for where to store additional information can vary based on the need. For Salesforce we know that developers may move the scaffolded models. When the designer was first run, the models were placed in Models\Salesforce\ folder. At some point someone refactored the project and moved the models to a shared portable library. The next time the developer attempts to update the consumption of Salesforce, the list of objects displayed as existing in the project are all unchecked as there aren’t any types in the Models\Salesforce folder. While not implemented in the Visual Studio 2015 Preview, we have a design where the developer can choose where the models currently exist. The developer clicks a browse button to view the solution finding the Models in the portable library. The tree is refreshed to reflect the models found. To meet the Great Memories goal, the provider will save off the last known location of the models in the .csproj file as a hintpath. By definition, this isn’t a turd as the .csproj file already has metadata in it. One could argue the .csproj file should only include build configuration information, but I’d argue turds need to go somewhere, and I’d rather see them in a bucket I know contains lots of stuff I don’t want to look at, and keep them out of my project directory.

# Summing it up

That sums up the experience we’ve targeted for consuming Connected Services, and the expectations for Connected Service Provider authors. We believe the experience will help developers be more productive with progressive discovery, learning, guiding them through and automating the tedious tasks by providing configuration over their services and service SDKs. We hope the new scaffolding and code ownership model is more productive than the “can’t touch this” model of the past. Lastly, we hope to continue evolving connected services with the reentrant/update scenarios to assure we’re delivering continual value over the life of your Visual Studio Development.