Coding the XSettings Type Provider

This is where life starts to get "interesting"! What you need to bear in mind is that, apart from some "tweaking" to do with Uri instances, all of our core code can be "ignored"; what we're interested in primarily from the core code is the assignment of the schema, the loading and validation of a users' XSettings file and the "map" that tells the type provider what settings to provide for and how to access their values.

A Type Provider is developed as an F# Class Library project. In the solution, `saTrilogy.TypeProviders`, I've created a new project named XSettings. The output assembly will be named `saTrilogy.TypeProviders.XSettings` and the "default" namespace I'll (manually) encode as `saTrilogy.TypeProviders`. There are now a number of steps we need to do to tell F# we're building a Type Provider and to set it up with the F# Type Provider Starter Pack from NuGet...

- Open **AssemblyInfo.fs** and add our new project as a friend assembly for the Common library using the expression...

```
[<assembly:InternalsVisibleTo("saTrilogy.TypeProvid↵
...  ers.XSettings")>]
```

- Save the changes to the Common library and rebuild the project.
- Add a project reference from the XSettings project to to the Common library. While you're at it, add `System.Xml` and `System.Xml.Linq` to the XSettings references as well.
- Open the NuGet package manager either for the XSettings project or for the solution against the XSettings project.
  - Do a search in the package manager against the `nuget.org` source for `Fsharp.TypeProviders.StarterPack` and install this package in the XSettings project.

  - I've kept the NuGet installed source file **DebugProvidedTypes.fs** in the project but, as it's not going to be used, you can delete it if you wish.
  - Ensure that the signature file **ProvidedTypes.fsi** is *above* the source file **ProvidedTypes.fs** in the solution explorer.
  - To eliminate a subsequent compilation warning open the Starter Pack **ProvidedTypes.fs** source and go to line #497. Change the `Seq.nth` expression to `Seq.item` (as the former is deprecated) then save and close the source file.

- In the **AssemblyInfo.fs** source file add the `open` declaration `open FSharp.Core.CompilerServices` and add the following expression...

```
[<assembly: TypeProviderAssembly>]
```

- Right click the project node in the solution explorer and select, from the context menu, the option **F# Power Tools ⇝ New Folder**. Name the new folder **Schema**.
- Right click the **Schema** folder and select **Add existing item** from the context menu. Add the **XSettings.xsd** schema we developed of which there is a copy in **LearningF#\Files\Schema**.
- For the **XSettings.xsd** file in the **Schema** folder, left-click it to select it in the solution explorer then go to the **Properties** tab for the selected item. For the **Properties** set the following...
  - **Bulid Action** to `EmbeddedResource`.
  - **Copy to Output Directory** to `Copy Always`.
- To the root of the project, add the existing file **Test.XSettings** from the **LearningF#\Files\Files** folder. For the **Test.XSettings** file, set the **Copy to Output Directory** to `Copy Always` in its properties from the solution explorer.

Now, for the specified **XSettings.xsd** schema, we intend to use this as the "default" schema for which I will specify three options - which, in order of selection preference refer to...

1. It will be sourced from a centralised web repository for which we'll have literals that identify the URL of the repository and the name of the schema.
2. It will be source from the host **bin** path (check the following section for a more thorough description of what I mean by "host") subdirectory named **Schema** since we've set the **Copy to Output Directory** value as `Copy Always`.

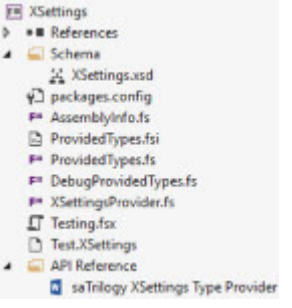

Figure 60: Layout of the XSettings Type Provider Project

3. If neither of the above work, then we can extract the schema from the Type Provider assembly as an embedded resource.

We actually want to allow the user to be able to specify a schema to use to validate their XSettings file but if they don't specify this parameter, (so it's clearly an optional parameter) then we need to provide a "default schema" - this is the purpose of the foregoing preference hierarchy; we'll use this cascading preference of the source of the XSettings schema as that against which we'll undertake the validation for a user specified XSettings file when the user does not specify a schema in invoking the Type Provider.

Everything we're going to add to the project will be placed after the Starter Pack source **ProvidedTypes.fs**. We need to implement the core code we've created and then we can implement the Type Provider.

Implementing XSettings

I'm implementing the Type Provider in the source file **XSettingsProvider.fs**; everything "else" we need is in the Common library - this source just implements the underlying data map for a user-requested source via the Common library. Firstly, lets' just take an overview of the source structure - I've removed all the comments that are present in the source and a lot of the "detail" in order to highlight the code sections that we're interested in...

```
namespace saTrilogy.TypeProviders

open System
open ProviderImplementation.ProvidedTypes
open Microsoft.FSharp.Core.CompilerServices
open System.Reflection

[<TypeProvider>]
type XSettings(config : TypeProviderConfig) as this =
  inherit TypeProviderForNamespaces()

  let XSettingsAssemblyName =
    "saTrilogy.TypeProviders.XSettings"

  let XSettingsSchemaName = "XSettings.xsd"

  let ProviderProjectPath = __SOURCE_DIRECTORY__

```

```
  let ns = "saTrilogy.TypeProviders"

  let asm = Assembly.GetExecutingAssembly()

  let settingsProvider =
    ProvidedTypeDefinition(
      asm, ns, "XSettings", Some typeof<obj>)

  let parameters =
    [ ProvidedStaticParameter("XSettingsFile",
                              typeof<string>)
      ProvidedStaticParameter("FileBase",
                              typeof<string>,
                              ProviderProjectPath)
      ProvidedStaticParameter("XSettingsSchema",
                              typeof<string>,
        DefaultSchemaPath
          WebRequestTimeout ProviderProjectPath
          XSettingsSchemaName XSettingsAssemblyName
          WebSchemaRepository)
      ProvidedStaticParameter("RequestTimeoutMillisecond↙
...   s",
                              typeof<int>,
                              WebRequestTimeout) ]

do settingsProvider.DefineStaticParameters
  ( parameters,
    fun typeName args ->
      let fileBase = args.[1] :?> string
      let timeout = args.[3] :?> int

      let settingsSchemaSetInfo = ...

      let settingsDocumentInfo = ...

      let provider =
        ProvidedTypeDefinition(
          asm, ns, typeName, Some typeof<obj>,
          HideObjectMethods = true)

      let targetSchemaProperty = ...

      let targetFileProperty = ...
```

```
62          let settingsMap =
              XSettingsMap settingsDocumentInfo.Document.Root
64
            for item in settingsMap do
66            let xid = item.Key
              let xSettingOf,xValue,xDelimiter,xComment =
68            item.Value.ElementName, item.Value.SettingValue,
              item.Value.Delimiter, item.Value.XmlComment
70
              let instProperty =
72              ProvidedProperty( ...
74          instProperty.IsStatic <- true
76          match xComment with
            | Some(value) -> instProperty.AddXmlDoc value
78          | None -> ()
80          provider.AddMember instProperty
82          provider )
84  do this.AddNamespace(ns, [settingsProvider])
```

Firstly, note the `namespace` - we set it to our base "default" namespace of `sa_Trilogy.TypeProviders`. There is no module declaration in this source since all we are declaring is a single type - named XSettings. This type name is how the user would "access" and declare their intent to use this Type Provider. This is a public type since we don't specify a visibility qualifier and we do, after all, want the user to actually "see" our Type Provider! The type is qualified by the attribute `[<TypeProvider>]` that is from the `FSharp.Core.CompilerServices` namespace.

There are a number of `open` statements - these are all pretty much "critical" for Type Provider implementation You'll note the absence of `FSharp.Reflection` as the reflection used by the Type Provider is limited to .NET reflection.

The syntax of the declaration of the XSettings type is crucial - apart from the `[<TypeProvider>]` attribute, we must...

1. Specify a class instantiation argument of `config : TypeProviderConfig`.
2. Specify an instantiation alias that we name `this`. The alias is used in the class instantiation primary `do` block and is required.
3. We must `inherit` from `TypeProviderForNamespaces` using its default constructor. That done, we define three crucial value types...

i. Named `ns` whose value will be the complete namespace to assign to the Type Provider - here `saTrilogy.TypeProvider`. The Type Provider is exposed as a "class" in this namespace.

ii. Named `asm` whose value is the "hosting" assembly - that which is declaring its intent to use this Type Provider.

iii. Named `settingsProvider` whose value is the output of an invocation to the function `ProvidedTypeDefinition` of the Starter Pack's **ProvidedTypes.fs** source. This value defines, for the sake of simplicity rather than "technical correctness", the "class" which is presented to the Type Provider user/host within which they'll "find" the provided types.

In our implementation the `settingsProvider` for the evaluated host assembly, has the specified namespace, and is named XSettings and will be presented as a .NET entity of type object. This is rather curious; we have named the implementation type, that is by default a public type in the `saTrilogy.TypeProviders` namespace to be XSettings (with an alias of `this`) and we have chosen the same name to be the implicitly public type provided to the host - in the same namespace! Surely we should expect a conflict? One's instinct is to name the implementation type to be different from the provided type but a "disadvantage" of this is that the developer would see, in referencing the assembly two public types regarding the Type Provider - it's implementation type and the provided type. I actually regard that as potentially confusing to the user of the Type Provider - "which one do they use" given that both will be shown in Intellisense? The jiggery pokery being undertaken by F# given the assembly attribute of `TypeProviderAssembly` in **AssemblyInfo.fs** actually enables us to name *both* the implementation and provided type as the **same** - then Intellisense will only present the user of the Type Provider with the provided type - the implementation type is, effectively, "hidden" - which I regard as a "good thing".

That said, our implementation class declares three private properties prior to the "mandatory" properties of `ns`, `asm` and `settingsProvider` - these are for "our" implementation requirements for the XSettings provider and details of their purpose is as follows...

Value Type	Purpose
`XSettingsAssemblyName`	If the user does not specify a schema to use with their settings file then we potentially may have to use an embedded resource for same - the value of this value type is the short name of the assembly that contains such an embedded resource.

Value Type	Purpose
`XSettingsSchemaName`	If the user does not specify a schema to use with their settings file then we have to provide one; the value of this value type is the name of the, thus default schema either present as a file of that name somewhere within a literal encoded web repository, the host project path, or the assembly with short name given by the value type `XSettingsAssemblyName`.
`ProviderProjectPath`	This is the `FileBase` used for searching for the default schema, if required. Recall that `__SOURCE_DIRECTORY__` is "context-sensitive"; as we are declaring its use in the implementation type then the source directory will be the project path of our XSettings project.

Type Provider Parameters

Most Type Providers require parameters that will govern from where their source the underlying data in the provision of types and their nested properties. The XSettings Type Provider has a potential requirement of four parameters...

`XSettingsFile` This is a reference to the XSettings file that the user wishes to expose via the Type Provider. Consequently, as a file reference, we'll assume it's a Uri so it can therefore be acceptable via a number of formats...

- As a Uri it can be a disk file, a disk folder (which would be erroneous) or a web file.
- A relative Uri - in which case we'll need a `FileBase` in order to locate the target.
- An absolute Uri for which, therefore, no `FileBase` is required.

`FileBase` If either the target XSettings file or the schema (which may be the default schema) is specified as a relative Uri then we need this `FileBase`, as an absolute Uri, in order to locate the referenced target relative Uri.

`XSettingsSchema` Every parsed XSettings file needs to be validated by a schema and in this parameter the user may target their own schema - if not, we'll use our default schema.

`RequestTimeoutMilliseconds` If a target Uri is web based then the user may want to override the literal encoded default timeout in the Common library with this value.

It's pretty clear from the foregoing that we can accommodate the schema parameter and timeout as optional parameters as we have mechanisms in the Common library whereby we can specify or dynamically evaluate default values or, by using our `ProviderProjectPath` of **__SOURCE_DIRECTORY__**, we can accommodate a default `FileBase`.

In our implementation we're using the `ProvidedStaticParameter`, which is declared as a class in the Starter Pack's `ProvidedType` source (it's easier to review in the signature file), to declare each of our parameters. In the code for the timeout parameter, the fourth element of a list of `ProvidedStaticParameter` assigned to the value type named `parameters`, we have...

```
ProvidedStaticParameter("RequestTimeoutMilliseconds",
...   typeof<int>, WebRequestTimeout)
```

The arguments required in providing an instance of a `ProvidedStaticParameter` are threefold...

1. The name to be given to the parameter.
2. The type to be associated with the parameter - here we've specified that the timeout must be an integer.
3. The value to use as a default in case the user should not provide one for the designated parameter - here we use, for example, the literal `WebRequestTimeout` from our Common library.

Video

260. Layout of the implementation class for a Type Provider - Parts I through III.

This, then, is how one specifies a parameter as optional - by simply specifying a default value in the implementation code. Note that the third argument of `ProvidedStaticParameter` is, itself, optional - one need not specify a default value for a parameter. Care needs to be taken in using defaults as regards the ordinal position of the parameter - one cannot make the second parameter of three optional when the third or the first is mandatory! We have specified our `FileBase` to be the second ordinal parameter as we require the name of the target XSettings file but we don't require the schema name. Consequently, for example, if the user were to specify an absolute Uri for both the XSettings target and schema then they must also include the second ordinal parameter of `FileBase` - for example, they may use their contextually sensitive **__SOURCE_DIRECTORY__** to reflect the project path from where *they* are invoking the Type Provider.

Another aspect of a parameter arises from the use of the word `Static` in the `ProvidedStaticParamter` class name; Type Provider parameters must be "static" in the same sense as we have used to date in dealing with literals; that is, their type must be a primitive type of the .NET Framework or their F# equivalent - so we cannot have a parameter whose type is a native F# type - such as a union,

record type, tuple etc. We have seen though, in our initial simple sample Type Provider, that we can use an `enum` member as a parameter - since the `enum` member is just an alias to an underlying integer (by default - because an `enum` can, for example, be defined as a set of unsigned short integers if required). In essence, whatever you wish to specify as a parameter must be evaluable by the F# compiler at design time (in the host assembly) - this is the restriction that is imposed in declaring literals and it is also thus imposed against Type Provider parameters.

Instantiation of a Type Provider Implementation

To recap to this point...

- Type Provider implementation is defined via a class declaration.
- We can name the implementation class to be the same as that of the Type Provider provided type - even in the same namespace.
- We have clear, mandatory requirements about how we declare the implementation class, it's inheritance and an alias for use during the instantiation of the class - `this`.
- we declare fields that define the namespace for the Type Provider and the host assembly (that which is invoking the Type Provider), then a value type as an instance of a `ProvidedTypeDefinition` class instance using these prior defined fields.
- We declare any parameters required and optional value defaults to be used in Type Provider instantiation. In an "ordinary" class, these would appear as instantiation arguments in the declaration of the type. For a Type Provider the instantiation argument of `config : TypeProviderConfig` is, shall we say, used internally by the Starter Pack to enable this indirect declaration of a provided type definition with a field exposing a list of parameters required by the type definition.

What happens next is the meat of our implementation; we have a `do` block that we know will be invoked as part of class instantiation. This do block has only one expression in it; it invokes your `settingsProvider` instance of a `ProvidedTypeDefinition` member named `DefineStaticParameters` (in the Starter Pack) and it uses, as its arguments...

1. The `parameters` list that a user provides as a field in the instantiation of the Type Provider implementation, and...
2. A lambda expression in which the anonymous function arguments are...
    i. The name of the Type Provider as a string and...
    ii. An object array, `obj[]`, whose elements will be the values of the

parameters that either the used specified or were used as default values.

The output of the lambda expression is an instance of a `ProvidedTypeDefinition`. Note that the output of the member `DefineStaticParameters` is `unit` - a do block can't "return" anything, it just undertakes code to complete the instantiation (the construction) of the class to which it belongs.

When a user, from a host assembly (or FSI), declares an intent to use the Type Provider by first specifying an `open` against its referenced namespace, your implementation code is dynamically compiled by F# within the host process. Assuming successful compilation, F# then awaits the expression that declares an intent to use the Type Provider - for example, the expression...

```
type tp = TPTest<"""U:\LearningF#\Code\TPTest\TPTest\file.txt""",Record.First>
```

As soon as this intent is intercepted - at design time (and at compile time), then F# uses the dynamically compiled Type Provider implementation to invoke an instantiation of the implementation class; then, for the above listing, the `do` block anonymous function argument of your instantiated `ProvidedTypeDefinition` instance member `DefineStaticParameters`, will receive as the name of the Type Provider `TPTest` and, for the object array, the value...

```
[ box """U:\LearningF#\Code\TPTest\TPTest\file.txt""",
  box Record.First ]
```

The intent of the body of the anonymous function is that of...

1. Parse the user-specified (or defaulted) arguments.
2. Declare *another* `ProvidedTypeDefinition`
3. Request the underlying source data and map it into provided types.
4. For each evaluated provided type append it to the newly declared `ProvidedTypeDefinition`.
5. Output the populated new `ProvidedTypeDefinition` to the invoker; this thus contains the provided types encapsulated by the Type Provider. Again, this is not the output of the `do` block but the output of the anonymous function - the second argument of the `DefineStaticParameters` evaluated by the `do` block.

As I've mentioned before, a `do` block cannot output anything other than `unit`; it's sole purpose is that of completing instantiation of a new class instance. Hence this jiggery pokery with the `DefineStaticParameters` member; it's not clear from the signature file how our "new" `ProvidedTypeDefinition` is going

to be used now that it encapsulates the provided types that are specified by the users' Type Provider parameters - whereas, the field `settingsProvider` is just a "blank slate". However, if we look at the source for `DefineStaticParameters` in the **ProvidedTypes.fs** file - as opposed to the signature file, what we see is...

```
member __.DefineStaticParameters
    (staticParameters : list<ProvidedStaticParameter>,
     apply : (string -> obj[] -> ProvidedTypeDefinition)) =
...
  staticParams      <- staticParameters
  staticParamsApply <- Some apply
```

It's clear from the code and, if you hover your mouse over the value type names of `staticParms` and `staticParmsApply`, that we're updating mutable value types with our "new" `ProvidedTypeDefinition`. So our evaluated provided types are thus encapsulated via the Starter Pack "ready for use" by the Type Provider invoker following completion of the implementation instantiation.

Before we consider these steps in detail, let me just cover the second "trivial" `do` block shown at the end of the implementation class - `do this.AddNamespace(ns, [settingsProvider])` - all this does is add our Type Provider namespace to the address space of the host assembly (or FSI) so that our work is then "visible". After all, the namespace of the host and that of the Type Provider need not be the same!

Parsing Type Provider Instantiation Parameters

The `fileBase` and `timeout` value types we accept at "face value"; they are just up-casted from the parameters object array, `args`, as, respectively, the second and fourth elements, to, respectively, a string and an integer. We cannot be so cavalier about the schema argument - the third argument of the incoming `args` object array. We must validate that the target file exists and is a valid Xml schema. The code for this, given our efforts in the Common library, is trivial...

```
let settingsSchemaSetInfo =
  MakeSchemaSet timeout
  <| Uri.Make(args.[2] :?> string, fileBase)
```

This makes use of the composite function `MakeSchemaSet` - a small variation of the previously considered `TrySchemaSet`, encoded as...

```
let MakeSchemaSet (timeout : int) =
  TryCompose.Initialise
  >> !^ (getSchemaStream timeout)
```

```
    >> !^ makeSchemaSet
    >> TryCompose.OutputValue
```

In our `settingsSchemaSetInfo` assignment we use `timeout` as the first, explicit argument; for the second, implicit argument of the composite function we evaluate a Uri and pipe the evaluation into the composite function invocation as the second argument. To evaluate the Uri we use our `Uri.Make` static extension - we give it as it's first argument, the schema target, the `args` object array third element up-casted to a string and, we use the parsed `fileBase` as the second argument to `Uri.Make`.

The `MakeSchemaSet` uses as its ultimate composite elemental the `TryCompose.OutputValue` function; hence, the composite function output will be a strongly typed instance of a `SchemaSetInfo` from the prior elemental or, if a failure occurred along the way, an appropriate exception will be raised - which would cause the Type Provider implementation to fall over in a heap with a **Type Initialization Exception** - since we're still in the process of instantiating the implementation class in its (primary) `do` block.

You should, in all this to do with the schema parameter, note that we cannot encode the default schema as a `SchemaSetInfo` instance for the Type Providers parameter; `SchemaSetInfo` is not a primitive type. Rather, for the default value of the parameter, we have encoded...

```
DefaultSchemaPath
  WebRequestTimeout
  ProviderProjectPath
  XSettingsSchemaName
  XSettingsAssemblyName
  WebSchemaRepository
```

This uses a number of our Common library literals as well as the previously defined fields to invoke the `DefaultSchemaPath` function whose output is the primitive type `string`. consequently, our default schema undergoes the same validation as would occur when the user specifies a string as a Type Provider input parameter for the schema. Of course, we need to take care of the fact that if our default schema is identified as the one that is encoded as an embedded resource, that we don't treat it as a Uri! That is handled by the elemental `defaultSchemaPath`.

Having evaluated `settingsSchemaPath`, for if it is not evaluated instantiation, by now, would have fallen over with a type initialisation exception, we can then consider the user specified parameter of the target XSettings file. We parse this argument and deal with it as follows...

```
let settingsDocumentInfo =
  XDocumentWithValidate
```

```
    timeout settingsSchemaSetInfo
    <| Uri.Make(args.[0] :?> string, fileBase)
```

Once again, a triviality given our Common library composite functions - it warrants no further mention.

Setting up the Modified ProvidedTypeDefinition

Now that we have a valid schema and a validated XSettings file against that schema, we can progress onto declaring our `ProvidedTypeDefinition` that will encapsulate our mapped values to be incorporated as provided types and thereby be "output" to the Type Provider host assembly.

We have used the value typed named `provider` as follows...

```
let provider =
  ProvidedTypeDefinition(
    asm, ns, typeName, Some typeof<obj>,
    HideObjectMethods = true)
```

We re-use the fields we previously declared for `asm` and `ns` - the host assembly reference and namespace for our Type Provider. For the `typeName` we will simply use the argument yielded by the anonymous function whose code body we're now processing.

We then have to specify the type for our Type Provider and bear in mind that even though we're dealing with an F# Type Provider - functionality not supported by other .NET languages, we are largely constrained by the use of .NET Framework defined types. The type we therefore specify for our Type Provider is `obj`! It's possible, at some much later stage, that you may want to present your Type Provider as a strongly-typed instance of some entity but, whatever that entity is, it must be .NET compatible - such as a class - it cannot be a native F# type. The matter of changing the type of the Type Provider to something other than `obj` is not one you'll want to address at this stage.

> **Video**
>
> 261. Parameter validation and type definition.

Now, that last argument - `HideObjectMethods`; this is actually a lot more useful that it may initially appear to be. It's best to think of this in terms of Intellisense and .NET; pretty much every entity in .NET inherits from `object` and `object` has a number of members - some of which like `ToString` are fairly useful! There are other `object` members - `Equals`, `GetHashCode` and `GetType`. Now, as we wish to type our Provided Type as an `obj`, that means that, for example, when we address it in code then Intellisense will "pick up" on these `obj` members and add them to its list. For a Type Provider its usually useful to not clutter the Intellisense "view" of your provided types with these `obj` members and, to enforce that requirement, one sets `HideObjectMethods = true` in the provided type definition.

Informational Provided Properties

Now, you may not ordinarily do this, but I want to provide two provided properties that qualify the source of the data from which the remaining provided properties are derived. I do this for two reasons...

1. The possibility of relative Uri addressing for none, one or two of the Type Provider parameters; consequently, I'll provide two string types that detail the absolute location of both the XSettings file and of the schema that was used to verify the XSettings file. This should then provide some feedback to the user that any relative Uri's that they've used with a `FileBase` (or the default one) are, in fact, the ones they wished to address - both for the XSettings target and the schema.
2. These are "small and simple" provided property definitions that characterise precisely what we need to do when it comes to providing properties for the encoded XSettings.

The code for these provided properties is as follows...

```
let targetSchemaProperty =
  ProvidedProperty(
    propertyName = "XSettingsSchema",
    propertyType = typeof<string>,
    GetterCode =
      ( fun _ ->
          let name = settingsSchemaSetInfo.Name
          <@@ name @@> ) )
targetSchemaProperty.IsStatic <- true
targetSchemaProperty.AddXmlDoc
  "The target XSettings Schema name."
provider.AddMember targetSchemaProperty

let targetFileProperty =
  ProvidedProperty(
    propertyName = "XSettingsFile",
    propertyType = typeof<string>,
    GetterCode =
      ( fun _ ->
          let name = settingsDocumentInfo.Name
          <@@ name @@> ) )
targetFileProperty.IsStatic <- true
targetFileProperty.AddXmlDoc
  "The target XSettings file name."
provider.AddMember targetFileProperty
```

The code for these two is essentially, functionally identical so we'll just con-

sider `targetSchemaProperty`: Firstly, we assign a value type as an instance of the Starter Pack `ProvidedProperty` class. The class constructor takes three arguments...

`propertyName` Of type `string`; this is the name of the provided property to be constructed. This will appear in the Intellisense pop-up as a property of the provided type - in our case, XSettings.

`propertyType` Of type `Type` - that is, a type specification; specifies the type of the property being constructed - for example. `typeof<string>`, `typeof<int>`, `typeof<float>` etc..

`parameters` An optional argument of type `list<ProvidedParameter>`; the constructor parses each element in the list and attempts to match the parameter specification against a named, public, settable property of the `ProvidedProperty` class. Theses properties are `GetterCode` and `SetterCode` (and that should raise some questions in your mind about what a Type Provider might be able to do!). We will restrict ourselves to the property named `GetterCode` - this is the expression used by the Type Provider to assign a value to the provided property. If we are only providing one `ProvidedParameter` then we do not have to use the usual list encapsulation of `[]` - as in our code - F# can figure this out on the fly.

Of these constructor arguments the provided property's name and type are self-evident in their purpose. Not necessarily so for the `GetterCode` - other than "in principle". This is where we "feed" the Type Provider with the value which it will dynamically assign to the provided property and this mechanism warrants further discussion. However, before doing so, having assigned our provided type let's quickly review the subsequent code, wherein...

1. We set the IsStatic property of the `ProvidedProperty` instance to `true`. We do this since we do not wish the user be required to declare a `new` instance of the `ProvidedType` before being able to access the `ProvidedProperty`. It is possible to create a `ProvidedType` that not only requires the user declare a new instance of it subsequent to the Type Provider "delivering" the provided type to the hosting application, but also permits the use of arguments for the new `ProvidedType` before being able to access its `ProvidedProperty` set. We don't consider such a Type Provider in this material.
2. We use the `AddXmlDoc` instance member to assign an Xml comment to the provided property that can then, immediately (Ok - usually with a "slight delay"!), be picked up by Intellisense in the host application at design time. There are a number of methods to assign such Xml comments - `AddXmlDoc` is the easiest "synchronous" method and that's the one we restrict ourselves to herein.
3. Having completed construction of the provided property and the assign-

ment/modification of any of *its* properties as required, we then append the provided property to the hosting provided type - in our code the provider `ProvidedTypeDefinition` instance. To do this we simply use the `AddMember` instance method of our provided type instance.

The foregoing is all rather straightforward except for the use of the provided property `GetterCode` - let's now consider this but, before doing so, take note...

With regard to `GetterCode`; "whatever it does", it does so synchronously at design time (or compile time) in the host application. It's important to remember that the consequence of the `GetterCode` is that whatever value is assigned thereby, it is the value that is encoded in IL into the host assembly so that, at runtime, this static value is all that the host assembly "sees". None of the intervening code we discussed is involved at host assembly runtime - its sole purpose is that, at design and compile time, this code is dynamically executed in order to embed a static, literal value into the host assembly.

Quotation Expressions in a Small Nutshell!

If you take a look at the declaration of GetterCode in the ProvidedTypes source or signature file we have...

```
member GetterCode : (Quotations.Expr list ->
...  Quotations.Expr)
  with set
```

This tells us that it's a mutable property that is assigned to by the provision of a function (because of the encapsulating `()` brackets) that has, as its input a list of `Quotation:Expr` and whose output is a *single* `Quotation:Expr`. To limit complexity we're not interested in the list aspect of this - we'll just, always use a single `Quotation:Expr`. Nevertheless, the question remains - what is `Quotation:Expr`?

Video

262. Defining informational provided properties - Parts I & II.

The trite answer is (along the lines that an `FSharpFunc` is a partially executed function) a `Quotation:Expr` is a "Quotation Expression". We have, to date, come across a variety of expressions such as the Linq Query Expression and the Lambda Expression. The characteristic of these, as far as the compiler is concerned, is that the expression is encapsulated by some character pair in its definition - for a Query that is `query{}` and for a lambda expression `()`. We're running out of bracketing characters here - in fact, in one way or another we've used all the "normal" ones, so a Quotation Expression is some expression that is delimited by the "brackets" `<>`! However,

just to further complicate things, there are two types of Quotation Expression! As follows...

1. An expression delimited by `<@@ @@>` is un-typed - it's output is, effectively, always `obj`.

2. An expression delimited by `<@ @>` is strongly typed; the output of the expression assumes the type of the evaluated, encapsulated expression. For example, for `<@ Math.Pow(2.,16.) @>`; the output of `Math.Pow` is of type `float` - thence that is the type assumed by the typed Quotation Expression and it will be expressed, in the signature as `Quotation:Expr<float>`.

I suppose the first thing you'd put your money on is that in Type Providers we use un-typed Quotation Expressions - after all, everything else to do with Type Providers is based upon .NET rather than native F# - so it probably also deals with objects rather than strongly typed values that would also permit inclusion of native F# types; you'd win your bet - you'll never see typed Quotation Expressions for a Type Provider.

That said, the question of what a Quotation Expression *is* remains unanswered. It helps I find, to cast one's mind back a few years to the earlier days of JavaScript (and some other languages); JavaScript had a keyword **eval** and, as one can now still use effectively (although, not necessarily, efficiently) in TSQL, you could build a string, say in a query, that "resembles" an SQL command - such as `Select column From table Where value = '{0}'` - not technically correct but you recognise `{0}` as a "place-holder" from our F# usage. You can then "dynamically" execute the string as a SQL command after replacing the parameter `{0}` with what you "need" *at that time* and, having dynamically built the command you can tell SQL to execute it as a SQL command and return its output value to you. For the JavaScript eval, it is the same principle - you wrap some string in the eval call and that string will be interpreted at runtime. The principle, therefore, is that of "*delayed evaluation of a value*". In that respect it's similar to the lazy keyword expect in the latter, one must type the *actual* expression required whereas, in the former, for a Quotation Expression, one can be more circumspect about *what* is to be lazily evaluated. I therefore regard the Quotation Expression as a "fuzzy", lazy expression!

Another difference between an F# Quotation Expression and the likes of the JavaScript eval and the TSQL dynamic SQL DML, is that the F# compiler will test your expression for syntactical correctness at design time - it will not wait until execution time to parse the expression and test it for syntactic correctness. An advantage of this is that, at design time, as long as the compiler is happy with your expression it can compile your assembly (even an "internal/dynamic FSI module") within which such is contained. Note, however, that as in dealing with .NET from F#, this does **not** guarantee that the evaluation of your expression will not cause an exception!

What the compiler *does* do, for a Quotation Expression - although I won't go into the details of what happens, is that, essentially, when the compiler parses a Quotation Expression, it builds an Abstract Syntax Tree (AST) of optimised code in IL that can be "directly" parsed and acted upon by the consumer of a quotation. Once again, I find the SQL analogy helps; in SQL a SQL Data Manipulation Language (DML) statement is written, by a developer, in "plain English" such as `Select column From table` but this "human-readable representation" of the command is meaningless to the SQL execution engine, consequently, the plain text is parsed by the SQL optimiser, checked for syntactical and logical validity and thence optimised into a tree of commands that the SQL runtime engine can "directly" parse as a sequence of directly executable steps it must undertake in order to satisfy the output requirement of the DML command. In coding our Quotation Expressions, the compiler creates the AST that it can then parse, in the host assembly at design or compile time (of the host assembly) and "inject" into the host assembly to "express" the value of the provided property. To my mind it is the complexity faced by the Starter Pack code in dealing with the un-typed Quotation Expression of the `GetterCode` that limits its capability to dealing only with primitive types but, as we've said, there's nearly always a work-around - as long as we always ensure our Quotation Expression is "acceptable" to the Starter Pack in yielding something "more useful" than a primitive type.

In simplistic terms we'll just summarise this by saying that a Quotation Expression is used to encapsulate syntactically correct code that is to be dynamically evaluated at "runtime" when the evaluation of the Quotation Expression (that, by then is an AST of IL expressions) is requested. This is technically correct in terms of the usage of the word runtime but, as always, you must remember - and I make no apology for reiterating the point, the following:

> For a Type Provider, "runtime" is actually design time and compile time of the host assembly - including FSI script. A Type Provider has no *real* "runtime" - such as an F# executable application does; the consequence of host assembly design/compile time usage of a Type Provider is simply the embedding of IL encoded static literals, or an AST to trigger instantiation of a type that only makes use of primitively typed literals as input, within the host assembly or dynamically generated FSI module.

The use of the Quotation Expression by the Type Provider has some interesting corollaries...

- It "doesn't matter" what expression you encode in the Quotation Expression - as long as it's syntactically correct the F# compiler will permit you to build your Type Provider assembly and thereby reference and use it from a host assembly.
- Because Type Providers use un-typed Quotation Expressions in their `GetterCode` then it "doesn't matter" what type of output it produces since

all un-typed Quotation Expressions are typed, in the best .NET tradition, as `object`.

Of course, we don't live in such an idealised world; the Type Provider Starter Pack makes damn sure that what you express as a Quotation Expression conforms to *its* requirements - and, even then, the probability of type initialisation exceptions occurring in the implementation instantiation code of your Type Provider during host assembly design/compile time may be high - there are some things the Starter Pack cannot pre-validate; it does what it can, but there are no guarantees that a compiled Type Provider assembly is flawless such that it can be guaranteed that there will never be any host assembly design/compile time type initialisation exceptions arising from the execution of an AST created by the `GetterCode` Quotation Expression.

One thing about type initialisation exceptions is that they are hard to debug at the best of times; herein one also has to contend with the thousands of lines of code in the Starter Pack and it's very easy for the cause of an exception to become obfuscated. Whilst we deal with debugging in the subsequent chapter, in order to ameliorate the issues about exception reporting, we "help" the Starter Pack enormously by having a consistent and thorough exception reporting mechanism in our Common library - especially via the use of our composite functions and their binding.

Assuming that, for a Type Provider, what we code in a Quotation Expression is syntactically correct so the compiler may create a valid AST tree of it in IL, just what does the Starter Pack require in terms of producing the value of a provided property? I'm not going to dig through the code but just highlight the requirements and corollaries...

- The output of the expression, of type obj, is the value that, in the host assembly, will be exposed as the value of the provided property.
- The `ProvidedProperty` instantiation argument of `propertyType` coerces the output of the `GetterCode` Quotation Expression to the "desired" type - so you better make sure these two match!
- The Starter Pack requires that the compiled AST of your `GetterCode` Quotation Expression **only** accepts input values that are typed as primitive types - regardless of the output type of the AST upon evaluation of the value of the provided property.

The "real trick" in using Type Providers is making use of that third requirement; the AST may only accept inputs that are primitive typed values but it can produce an output of type `obj` - which encompasses .NET and user-defined classes! However, great care needs to be taken when coding the `GetterCode`; as a Quotation Expression you can use any F# expression even including native F# types and the F# compiler *will* accept that - as long as it's a syntactically valid expression whose output is of type `obj`. Your Type Provider (notwithstanding any other

errors) will compile and you can attempt to use it from a host assembly. However, at that time, the design/compile time of the host assembly referencing your Type Provider, the Starter Pack code will fail as part of the implementation classes instantiation and all you'll see is a type initialisation exception. This is such an important point that I'm going to highlight and reiterate it...

> For a Type Provider, in your `GetterCode` Quotation Expression for provided properties, the compiled Quotation Expression AST can **only** accept input values in evaluating the `obj` typed output of the AST where the input values have a type that is a primitive .NET type - or their F# equivalent - for example, the .NET `Double` is the same as an F# `float`.

For our informational provided properties of `targetSchemaProperty` and `targetFileProperty`, our Quotation Expressions are trivial - it is just the name of a value type that has, "prior" to the Quotation Expression declaration, been evaluated with a primitive type of string - as in the code...

```
GetterCode =
  ( fun _ ->
      let name = settingsDocumentInfo.Name
      <@@ name @@> ) )
```

The `GetterCode` is a an anonymous function, whose input argument (which, according to the Starter Pack will be a value type with a type of `list<Quotation.Expr>`) we're not interested in, produces as its output a single, un-typed `Quotation.Expr`. In the anonymous function we declare a value type evaluated as the `Name` property of either the `SchemSetInfo struct` or the `XDocumentInfo struct` in the Common library module `InternalTypes`, we then use the value type as the subject of a Quotation Expression to be output to the `ProvidedProperty` instance.

Thus, for our informational provided properties, this is reasonably straightforward. However, for the XSettings provided properties, let alone those we'll deal with later for our other Type Providers, this can become more complex but, if you bear the previously highlighted "rules" in mind, you should not have a problem - especially if, in the anonymous function of the `GetterCode`, you "massage" your data into primitive types that can be easily expressed in the Quotation Expression. .

Mapping Underlying Data to Provided Properties

Now we've seen how it's simplistically done - that is, the creation of a provided property, we need to consider exposing the XSettings values in the user specified XSettings file as provided properties.

Of course, the first thing we need to do is to go to the Common library and, for the user-specified and validated XSettings file, construct a map of the under-

lying data that will quantify what provided properties we need to cater for. The code for this is...

```
let settingsMap =
  XSettingsMap settingsDocumentInfo.Document.Root
```

The `XSettingsMap` function and its supporting code, in the Common library internal module XSettings, is defined as follows...

```
type XSettingsNodeInfo =
  struct
    val ElementName : string
    val SettingValue : string
    val Delimiter : char
    val XmlComment : string option

    new(elementName,value,delimiter,xmlComment) =
      { ElementName = elementName; SettingValue = value;
        Delimiter = delimiter; XmlComment = xmlComment }
  end

let private xNameXid = XName.Get("XId")
let private xNameComment = XName.Get("Comment")
let private xNameDelimiter = XName.Get("Delimiter")

let XSettingsMap (rootElement : XElement) =
  rootElement.Elements()
    .Select(
      fun xe ->
        let xid = xe.Attribute(xNameXid).Value
        xid,
        XSettingsNodeInfo(
          xe.Name.LocalName,
          xe.Value,
          ( match xe.Attribute(xNameDelimiter) with
            | null -> Unchecked.defaultof<char>
            | attr -> attr.Value.ToCharArray().[0] ),
          ( match xe.Attribute(xNameComment) with
            | null -> None
            | attr -> Some(attr.Value) ) ) )
  |> Map.ofSeq
```

We declare a `struct` named `XSettingsNodeInfo` with the fields...

`ElementName` The name of the element pertaining to the setting - such as `String`, `StringArray` etc. The name will be used as a “switch” in order to specify the output type of the provided property.

`SettingValue` The value of the setting as expressed in the Xml XSettings file. All element values are of type string or Base64Binary. In the event of the latter, we'll use .NET facilities to dynamically convert a Base64Binary value into a string value or a byte array, as required.

`Delimiter` For array based setting this identifies the (single) character that will be used in a .NET `Split` of the input string into a .NET `Array`. We will then type the array as appropriate to the parsed `ElementName`.

`XmlComment` The Xml encoded (optional) string that will serve as an Xml Intellisense comment for the provided property.

The `struct` has a `new` declaration whereby we require, for construction of an `XSettingsNodeInfo` instance, arguments that represent, in ordinal order, each of the field values to be assigned to the `struct` fields.

We then declare three value types whose values are the Xml Linq XName instances of the attributes that we wish to extract for a settings' element.

Now we can consider the function that maps the underlying data into an F# map of type `Map<string,XSettingsNodeInfo>` where the map key, of type string, is the `XId` attribute value of the setting and the value is an `XSettingsNodeInfo` instance that encapsulates all the information we need with which to construct the provided property. The `XSettingsMap` function argument is the root node of the user-specified XSettings file, for which...

- We invoke the Xml Elements instance member of XElement to fetch all of the child elements of the root element into an IEnumerable collection.
- We use a Linq `Select` to map the IEnumerable collection such that, for each element of the collection, in turn...
  - We evaluate the value type `xid` to be the value of the elements' `XId` attribute.
  - We construct a tuple with the two elements...
    1. The `xid` value.
    2. A new instance of an `XSettingsNodeInfo` structure, such that...

       `ElementName` Is assigned as the XName `LocalName` string of the element name.

       `SettingValue` Is assigned as the element value.

       `Delimiter` The `Delimiter` attribute value is parsed through a pattern match and assigned as the first element of the `ToCharArray` representation of the incoming string value or, if not present, the unchecked default of the `char` type.

       `XmlComment` The `Comment` attribute value is parsed through a pattern match and assigned as the string input if it exists as an option type otherwise as None.

- The IEnumerable collection output by `Select` is then converted to an F# map via `Map.ofSeq`.

We can now iterate through this map such that, for each map element, we can "convert" its key/value pair into a provided property. In processing the map we iterate through each key/value pair as follows - where I've removed the majority of the property assignments to make the whole clearer...

```
for item in settingsMap do
  let xid = item.Key

  let xSettingOf,xValue,xDelimiter,xComment =
    item.Value.ElementName, item.Value.SettingValue,
    item.Value.Delimiter, item.Value.XmlComment

  let instProperty =
    ProvidedProperty
      ( propertyName = xid,
        propertyType =
          ( match xSettingOf with
            | "String" -> typeof<string>
            | "StringArray" -> typeof<string[]>
            // The rest of the cases here...
            | "ByteArray" -> typeof<byte[]>
            | value ->
                failwith
                <| "Unknown Setting element of " + value
          ),
        GetterCode =
          ( fun args ->
              match xSettingOf with
              | "String" ->
                  <@@ xValue @@>
              | "StringArray" ->
                  <@@ xValue.Split(xDelimiter) @@>
              // The rest of the cases here...
              | "ByteArray" ->
                  <@@ Convert.FromBase64String xValue @@>
              | value ->
                  failwith
                  <| "Unknown Setting element of " + value
          )
      )

  instProperty.IsStatic <- true
```

```
38
        match xComment with
40      | Some(value) -> instProperty.AddXmlDoc value
        | None -> ()
42
        provider.AddMember instProperty
```

This is straightforward; for every `item` in the map...

- Assign the value type `xid` to be the key of the map.
- Decompose the value structure into four distinct value types named `xSettingOf`, `xValue`, `xDelimiter` and `xComment` respectively. These are the value types we'll use in assigning the provided property.
- Declare a `new ProvidedProperty` instance named `instProperty` with the instantiation arguments...

  `propertyName` Assigned as the value of `xid`.

  `propertyType` We use a pattern match against the XName property of `LocalName` and for each "expected" element name we assign a `typeof<>` using the type we wish to associate with the setting value.

  - For the singleton settings this is simple enough - it's just `string`, `int`, `float` etc.
  - For the array based settings values we use the type as for its matching singleton but just suffix the value type name with `[]` to signify it as an array of the specified type.
  - For the `ByteArray` setting we use a byte array as in `byte[]`.

  For any element names we don't list as pattern match cases we will terminate instantiation processing using a `failwith` - this will, in turn, raise a type initialisation exception.

  `GetterCode` Within the anonymous function of the `GetterCode` we again use a pattern match so processing is discriminated by the type of the setting we are dealing with for each incoming `item`. The Quotation Expression is encoded on the basis of the setting value - the value of the value type named `xValue` and we then undertake type conversions so that, at design/compile time of the AST of the compiled Quotation Expression is evaluated with the type as expressed via the `propertyType` value type. We undertake the following type conversions in the expression of the Quotation Expression - and, bear in mind, our "input" type is always a primitive type of string being that of the value type `xValue`...

Element Name	Quotation Expression for Type Conversion
`String`	`xValue`

Element Name	Quotation Expression for Type Conversion
StringArray	`xValue.Split(xDelimiter)`
Char	`Char.Parse(xValue.Substring(0,1))`
CharArray	`[| for txt in xValue.Split(xDelimiter) do match txt.Length with | x when x = 1 -> let chrs = txt.ToCharArray() yield chrs.[0] | _ -> () |]` This is a "tricky" one; after we've split the Xml string into an array of string, then, for each element of the array we have to take the first element of the `ToCharArray` of the element string - if we can!
IsoDate	`IsoDate.Parse xValue`
IsoDateArray	`[| for txt in xValue.Split(xDelimiter) -> IsoDate.Parse txt |]` You'll note the existence of the static `Parse` method we added to our `IsoDate` user-defined class to handle this parsing of an ISO formatted date string from Xml.
IsoTime	`IsoTime.Parse xValue`
IsoTimeArray	`[| for txt in xValue.Split(xDelimiter) -> IsoTime.Parse txt |]` As for `IsoDate`, we have a static `Parse` method for our `IsoTime` user-defined class.
DateTime	`DateTime.Parse xValue`
DateTimeArray	`[| for txt in xValue.Split(xDelimiter) -> DateTime.Parse txt |]` We used the .NET `DateTime.Parse` similarly as for how we created the `Parse` method for our own `IsoDate` and `IsoTime` classes.
Short	`Int16.Parse xValue`

Element Name	Quotation Expression for Type Conversion
ShortArray	`[| for txt in xValue.Split(xDelimiter) -> Int16.Parse txt |]` Note our use of the .NET `Int16` member `Parse` - F# will "read" the .NET type as a "native" `int16`. We use this principle for all of the other numeric types following - as in the .NET method that F# then interprets as the "required" F# type.
ShortUnsigned	`UInt16.Parse xValue`
ShortUnsigned Array	`[| for txt in xValue.Split(xDelimiter) -> UInt16.Parse txt |]` The F# native type of the output is `uint16[]`.
Int	`Int32.Parse xValue`
IntArray	`[| for txt in xValue.Split(xDelimiter) -> Int32.Parse txt |]` The F# native type of the output is `int[]`.
IntUnsigned	`UInt32.Parse xValue`
IntUnsigned Array	`[| for txt in xValue.Split(xDelimiter) -> UInt32.Parse txt |]` The F# native type of the output is `uint[]`.
Long	`Int64.Parse xValue`
LongArray	`[| for txt in xValue.Split(xDelimiter) -> Int64.Parse txt |]` The F# native type of the output is `int64[]`.
LongUnsigned	`UInt64.Parse xValue`
LongUnsigned Array	`[| for txt in xValue.Split(xDelimiter) -> UInt64.Parse txt |]` The F# native type of the output is `uint64[]`.
Float32	`Single.Parse xValue |> float32`

Element Name	Quotation Expression for Type Conversion
Float32Array	`[| for txt in xValue.Split(xDelimiter) -> Single.Parse txt |> float32 |]` The .NET `Single` type is synonymous with the F# `float32`.
Float	`Double.Parse xValue |> float`
FloatArray	`[| for txt in xValue.Split(xDelimiter) -> Double.Parse txt |> float |]` The .NET `Double` type is synonymous with the F# `float`.
ByteArray	`Convert.FromBase64String xValue` We are using the .NET `Convert` class to convert the incoming string - which is Xml encoded as a Base64Binary string into a byte array.

As for the `propertyType`, if we encounter an "unexpected" case for the pattern match, then we'll terminate initialisation by issuing a `failwith`.

- Having created a provided property instance we then set its `IsStatic` value to `true`, assign an Xml comment if we can and append the newly created provided property to our provided type - the value type named `provider`.

After the `for` iteration through our map of settings values we just encode `provider` - so our now populated provided type is the desired output of the `settingsProvider.DefineStaticParameters` argument of its lambda expression. As we've seen, this is then used to update the mutable property in the Starter Pack that exposes the provided type to the host assembly.

For the `GetterCode`, note the following...

- You can see there is no restriction in the Quotation Expression against the invocation of functions and members - either from the .NET framework or elsewhere as long as only primitive types are used in their invocation as input.
- This means that we can "model" output types for our type providers under the proviso that these types can be instantiated solely by primitive types. We have used two such "models" for the XSettings provider - for `IsoDate` and `IsoTime` and we use the .NET `DateTime` for another setting since

it, too, has a member that can output an instance of itself given only primitive types to work with.

Video

264. Evaluating Provided Properties for the XSettings Type Provider - Parts I through III.

The `string` type, as a primitive type, is pretty "uninteresting" even though we use it to good effect here. What took me a while to appreciate is that another primitive type is `byte` and, as with all primitive types, that means that a byte array - `byte[]`, is *also* a primitive type. In the future, you'll find that we spend a lot more time and effort in dealing with byte arrays for our Quotation Expressions.

As we've already stated that the second `do` block of the implementation class just covers the adding of the Type Provider namespace to the address space of the host assembly, this completes the instantiation of the implementation of the Type Provider. Control flow, via the Starter Pack, will now return to the host assembly where one can make use of the instantiated Type Provider.