F#. Type Providers

Korzystanie z danych w różnych formatach, z wielu źródeł, jest obecnie standardowym zadaniem nowoczesnego oprogramowania. Jest to jednocześnie wyzwanie dla programistów. Naszym zadaniem jest pozyskanie tych danych oraz stworzenie pomostu między zewnętrznym źródłem danych a domeną naszego systemu. W tym artykule postaram się przybliżyć komponenty type providers w języku F# – narzędziami, które ułatwią nam pracę z danymi o różnych formatach i pochodzeniu.

Możliwych scenariuszy jest wiele. Dane mogą pochodzić z różnych baz danych, z plików, z usług sieciowych. Dodatkowo każde z tych źródeł może narzucić swój format danych. Dostawcy typów (ang. *type providers*) potrafią w łatwy sposób pozyskać dane z bazy danych czy usługi REST. Dane zostaną dostarczone wraz z typami specyficznymi dla danego źródła danych. Typy będą dokładnym odwzorowaniem struktur danych, które pozyskujemy, a ich stworzeniem zajmie się właśnie dostawca typu. Tutaj widać główną zaletę tego podejścia – jesteśmy zwolnieni z tworzenia kodu pośredniego, za pomocą którego zewnętrzne dane byłyby mapowane na naszą domenę. Mając narzędzie w postaci dostawców typów, możemy w łatwy sposób łączyć dane z wielu źródeł, migrować rekordy między różnymi bazami danych czy realizować scenariusz ETL (Extract / Transform / Load) za pomocą skryptu w F#. Zanim przejdziemy do przykładów, parę słów o mechanizmie działania dostawców typów.

Dostawca typu ma za zadanie wykonanie następujących operacji: połączenie się ze źródłem danych, rozpoznanie struktury danych, przygotowanie typów dla tej struktury oraz umożliwienie dostępu do danych. Cały ten proces odbywa się podczas kompilacji programu i jest dla nas prawie niezauważalny. Przypomina to nieco automatyczne generowanie kodu na podstawie udostępnionego schematu, na przykład WSDL. Jednak proces ten odbywa się automatycznie oraz, w przeciwieństwie do generatorów kodu (takich jak generator klasy proxy dla usługi sieciowej WCF), jest dynamiczny – każda zmiana struktury powoduje wygenerowanie nowych wersji typów. Nie jesteśmy narażeni na sytuację, w której nasz kod używałby nieaktualnych typów. Zostaną one automatycznie odświeżone po zmianie wewnątrz źródła danych, a nasz kod prawdopodobnie przestanie się kompilować, co wymusi na nas natychmiastową jego poprawę. Dodatkowo dostawcy typów są silnie powiązani z kompilatorem F# oraz IDE – z Visual Studio. Oprócz samych typów dostawcy często udostępniają dokumentację do nich, co jest bardzo pomocne podczas pracy w IDE – w Visual Studio eksplorujemy typy dzięki IntelliSense, tak jakbyśmy pracowali z typami z własnej domeny aplikacji.

Dostawcy typów ułatwiają pracę z danymi, jednak same w sobie nie są proste do zbudowania. Do dynamicznej budowy typów wykorzystują cytowania (ang. *quotations*) – zaawansowaną technikę, dostępną w języku F#. W tym tekście przede wszystkim skupię się na przykładach użycia najpopularniejszych bibliotek, które realizują najczęściej spotykane potrzeby (dostęp do usług REST i baz danych). Następnie przybliżę sposób tworzenie własnego, prostego type provider'a. Zainteresowanych tworzeniem dostawców typów odsyłam do artykułu „Creating a Type Provider" (https://goo.gl/Ytpx0m) oraz do kursu „Building F# Type Providers" (https://goo.gl/oU2vLJ).

Kim są dostawcy z biblioteki FSharp.Data (http://fsharp.github.io/FSharp.Data/)? Biblioteka ta jest tworzona przez społeczność związaną z F# Software Foundation (http://fsharp.org/). Jest to środowisko rozwijające język F# oraz jego biblioteki, które ma obecnie największy wkład w rozwój i popularyzację tego języka. FSharp.Data, oprócz dostawców typów, zawiera szereg innych komponentów ułatwiających pracę z formatami HTML, JSON oraz komunikacją HTTP. Jednak nas interesować będą następujące komponenty dostępne w FSharp.Data: CSV Type Provider, HTML Type Provider, JSON Type Provider, XML Type Provider oraz WorldBank Provider. Następnie przedstawię kolejne biblioteki i dostawców pracujących z bazami danych: FSharp.Data.TypeProviders, FSharp.Data.SqlClient oraz FSharp.Data.SQLProvider. Ok, czas na trochę kodu (każdy z przykładów będzie prezentowany w formie niewielkiej aplikacji konsolowej utworzonej w Visual Studio 2015).

FSHARP.DATA CSV TYPE PROVIDER

http://fsharp.github.io/FSharp.Data/library/CsvProvider.html

CSV Type Provider umożliwia łatwy dostęp do danych w plikach CSV. W przykładzie posłużę się prostym plikiem *Persons.csv* z trzema kolumnami: `Id`, `Name` i `Salary`.

Listing 1. Plik Persons.csv

```
Id;Name;Salary
1;Bob;1500.56
2;Alice;2455.23
3;John;1890.97
```

Listing 2. Przykład użycia CsvProvider

```
open System
open FSharp.Data

type Persons = CsvProvider<"C:\Persons.csv",";">

[<EntryPoint>]
let main argv =
  let persons = Persons.Load("C:\Persons.csv")
  persons.Headers.Value |> Seq.iter (fun header -> printf "%-
  10s" header)
  printfn ""
  persons.Rows |> Seq.iter (fun row -> printfn "%-10i%-10s%-10f"
  row.Id row.Name row.Salary)
  Console.ReadLine() |> ignore
  0
```

Ten krótki program wczytał i zaprezentował dane z pliku CSV. Zwróćmy uwagę na linię, w której został utworzony typ `Persons`. To właśnie tutaj dostawca stworzył potrzebne typy do komunikacji ze źródłem danych. Jedynymi parametrami są ścieżka do pliku oraz opcjonalny separator. Do kolumn odwołujemy się za pomocą

kolekcji `Headers`, a do wierszy – `Rows`. Co istotne, dane z kolumn otrzymały odpowiednie typy: `Id – int`, `Name – string`, `Salary – decimal`.

FSHARP.DATA HTML TYPE PROVIDER

http://fsharp.github.io/FSharp.Data/library/HtmlProvider.html

HTML Type Provider ułatwia przetwarzanie danych w postaci HTML. W przykładzie użycia tego dostawcy wykorzystamy dane z Wikipedii, dotyczące najlepszych strzelców w piłkarskiej Lidze Mistrzów.

Listing 3. Przykład użycia HtmlProvider

```
open System
open FSharp.Data

[<Literal>]
let championsLeagueUrl =
  "https://en.wikipedia.org/wiki/
  List_of_European_Cup_and_UEFA_Champions_League_top_scorers"

  type ChampionsLeague = HtmlProvider<championsLeagueUrl>

[<EntryPoint>]
let main argv =
  let championsLeague = ChampionsLeague.Load(championsLeagueUrl)
  printfn "%-5s%-25s%-5s" "No." "Player" "Goals"
  championsLeague.Tables."All-time top scorers".Rows
  |> Seq.take 20
  |> Seq.iteri (fun i row -> printfn "%-5i%-25s%-5s" (i + 1)
  row.Player (row.Goals.Substring(0, 2)))
  Console.ReadLine() |> ignore
  0
```

Dostawca wyodrębnił z dokumentu HTML wszystkie tabele (zawartość elementów `table`). Nazwa kolekcji jest generowana na podstawie atrybutu tabeli HTML'owej – `id`, `name`, `title`, `summary` lub `caption`. Jeżeli żaden z nich nie jest uzupełniony, kolekcja otrzymuje nazwę domyślną. Dane, które chcieliśmy pozyskać, są dostępne w kolekcji `"All-time top scorers"`. Iterując po wierszach kolekcji, wybieramy odpowiednie property (zgodne z nazwami header'ów tabeli HTML), a program wyświetla oczekiwany wynik.

FSHARP.DATA JSON TYPE PROVIDER

http://fsharp.github.io/FSharp.Data/library/JsonProvider.html

JSON Type Provider to dostawca typów na podstawie źródła w formacie JSON. Jako źródło może posłużyć ciąg znaków reprezentujący dane w formacie JSON, plik lub zasób sieciowy czy wynik wywołania restowego API.

Listing 4. Przykład użycia JsonProvider – parsowanie ciągu znaków

```
open System
open FSharp.Data

type Person = JsonProvider<""" { "Name": "name",
                                 "Address":
                               { "City": "city",
                                 "Street": "street" }
                               } """>

[<EntryPoint>]
let main argv =
    let bob = Person.Parse(""" { "Name": "Bob",
                                 "Address":
                               { "City": "New Port Richey",
                                 "Street": "85 Bay Drive" }
                               } """)
  printfn "%s %s %s" bob.Name bob.Address.City bob.Address.
  Street
  Console.ReadLine() |> ignore
  0
```

Listing 5. Przykład użycia JsonProvider – pobranie danych z API

```
open System
open FSharp.Data

[<Literal>]
let jsonUrl = "http://mysafeinfo.com/api/data?list=planets&format=json"

type Planets = JsonProvider<jsonUrl>

[<EntryPoint>]
let main argv =
  let planets = Planets.Load(jsonUrl)
  printfn "%-10s%-5s" "Name" "Description"
  planets |> Seq.iter (fun row -> printfn "%-10s%-5s" row.Nm
  row.Gen)
  Console.ReadLine() |> ignore
  0
```

W obu przykładach struktura dokumentu JSON została przełożona na odpowiednie typy w F#. W pierwszym przykładzie podaliśmy szablon danych, a następnie wykonaliśmy parsowanie ciągu znaków zgodnych ze strukturą. W drugim przykładzie odwołaliśmy się do danych z API i ponownie otrzymaliśmy typ zgodny ze strukturą danych zwracanych przez źródło.

FSHARP.DATA XML TYPE PROVIDER

http://fsharp.github.io/FSharp.Data/library/XmlProvider.html

XML Type Provider jest bardzo podobny w użyciu do JSON Type Provider. Przykłady użycia będą analogiczne, z tą różnicą, że będziemy konsumować dane w formacie XML.

Listing 6. Przykład użycia XmlProvider – parsowanie ciągu znaków

```
open System
open FSharp.Data


type Persons = XmlProvider<""" <person name="name" age="0" />
""">

[<EntryPoint>]
let main argv =
  let bob = Persons.Parse(""" <person name="Bob" age="35" /> """)
  printfn "%-10s%-10i" bob.Name bob.Age
  Console.ReadLine() |> ignore
  0
```

Listing 7. Przykład użycia XmlProvider – pobranie danych z API

```
open System
open FSharp.Data

[<Literal>]
let xmlUrl = "http://mysafeinfo.com/api/data?list=planets&format=xml"

type Planets = XmlProvider<xmlUrl>

[<EntryPoint>]
let main argv =
  let planets = Planets.Load(xmlUrl)
  printfn "%-10s%-5s" "Name" "Description"
  planets.Ps |> Seq.iter (fun row -> printfn "%-10s%-5s" row.
  Nm row.Gen)
  Console.ReadLine() |> ignore
  0
```

Podobnie jak w przypadku JsonProvider, w pierwszym przykładzie parsowany jest ciąg znaków, a w kolejnym dane z API.

FSHARP.DATA WORLDBANK PROVIDER

http://fsharp.github.io/FSharp.Data/library/WorldBank.html

WorldBank Provider udostępnia szereg danych finansowych i demograficznych z wielu krajów. Więcej informacji o World Bank Open Data znajduje się pod adresem: http://data.worldbank.org/.

Listing 8. Przykład użycia WorldBank Provider

```
open System
open FSharp.Data
[<EntryPoint>]
let main argv =
  let data = WorldBankData.GetDataContext()

  data.Regions."European Union".Countries
  |> Seq.iter (fun x -> printfn "%s %s" x.Name x.CapitalCity)

  printfn "%f" (data.Countries.Poland.Indicators.''Population,
  total''.Item 2015)

  Console.ReadLine() |> ignore
  0
```

FSHARP.TEXT.REGEXPROVIDER

http://fsprojects.github.io/FSharp.Text.RegexProvider/

RegexProvider jest nieco inny od dostawców, których zaprezentowałem do tej pory. Operuje on na wyrażeniu regularnym i udostępnia właściwości, które odpowiadają grupom nazwanym tego wyrażenia, co może ułatwić pracę z przetwarzaniem tekstu za pomocą wyrażeń regularnych.

Listing 9. Przykład użycia RegexProvider

```
open System
open FSharp.Text.RegexProvider
type DateRegex = Regex< @"(?<Year>\d{4})-(?<Month>\d{2})-(?<Day>\d{2})" >
[<EntryPoint>]
let main argv =
  let m = DateRegex().TypedMatch("2017-02-20")
  printfn "%s" m.Year.Value
  "Press enter to exit." |> Console.WriteLine
  0
```

To dość nietypowy dostawca typu – nie bazuje na danych tylko na innym typie.

FSHARP.DATA.TYPEPROVIDERS

https://docs.microsoft.com/pl-pl/dotnet/articles/fsharp/tutorials/type-providers/accessing-a-sql-database

Pora na przedstawienie dostawców typów związanych z bazami danych. SqlDataConnection umożliwia dostęp do baz MSSQL. Za pomocą tego dostawcy możemy wykonywać między innymi następujące operacje: zapis i odczyt danych, wywoływanie procedur składowanych oraz modyfikację struktur bazy danych. W przykładzie posłużę się bazą MSSQL LocalDB, na której zdefiniowana będzie prosta tablica `Items`, z dwoma kolumnami – `Id` i `Name`.

Listing 10. Przykład użycia SqlDataConnection

```
open System
open FSharp.Data.TypeProviders

type DB = SqlDataConnection<"Data Source=(localdb)\
MSSQLLocalDB;Initial Catalog=HelloWorld;Integrated
Security=True;">

[<EntryPoint>]
let main argv =

  let ctx = DB.GetDataContext()

  let row1 = new DB.ServiceTypes.Items(Id = 1, Name = "Foo")
  let row2 = new DB.ServiceTypes.Items(Id = 2, Name = "Bar")
  let row3 = new DB.ServiceTypes.Items(Id = 3, Name = "FooBar")

  ctx.Items.InsertAllOnSubmit([row1; row2; row3])

  ctx.DataContext.SubmitChanges()

  let query = query {
    for row in ctx.Items do
    select row
    }
  query |> Seq.iter (fun row -> printfn "%i %s" row.Id row.Name)

  Console.ReadLine() |> ignore
  0
```

Dostawca SqlDataConnection współpracuje z bazami MS SQL Server. Automatycznie tworzy typy DTO na podstawie struktury bazy. Umożliwia wykonywanie operacji CRUD w sposób podobny jak przy wykorzystaniu bibliotek ORM. Do zapisu danych używamy wygenerowanych funkcji, tworzących wiersze danej tabeli, które po dodaniu do kolekcji `Items` utrwalamy metodą `SubmitChanges` z bieżącego DataContext. Zapytania wykonujemy poprzez query expressions – konstrukcję składniowo podobną do Linq z C#.

FSHARP.DATA.SQLPROVIDER

http://fsprojects.github.io/SQLProvider/

Kolejny dostawca – SqlProvider – również współpracuje z bazami danych. O ile poprzedni dostawca typów umożliwia korzystanie jedynie z baz MSSQL, to SqlProvider daje dostęp do wielu serwerów baz danych (między innymi: MSSQL, MySQL, Oracle, PostgreSQL). Praca z SqlProvider jest podobna do SqlDataConnection, jednak to obsługa wielu serwerów baz danych daje ogromne możliwości i szerokie zastosowanie tego komponentu. Przede wszystkim, w łatwy sposób, możemy migrować dane między różnymi bazami. W przykładzie ponownie posłużę się tabelą `Items`, wykorzystaną w kodzie z Listingu 10.

Listing 11. Przykład użycia SqlProvider

```
open System
open FSharp.Data.Sql

[<Literal>]
let connectionString = "Data Source=(localdb)\
MSSQLLocalDB;Initial Catalog=HelloWorld;Integrated
Security=True;"

type DB = SqlDataProvider<
   ConnectionString = connectionString,
   DatabaseVendor = Common.DatabaseProviderTypes.MSSQLSERVER>

[<EntryPoint>]
let main argv =

  let ctx = DB.GetDataContext()

  let row1 = ctx.Dbo.Items.''Create(Id, Name)''(1, "Foo")
  let row2 = ctx.Dbo.Items.''Create(Id, Name)''(2, "Bar")
  let row3 = ctx.Dbo.Items.''Create(Id, Name)''(3, "FooBar")

  ctx.SubmitUpdates()

  let q = query {
    for row in ctx.Dbo.Items do
    sortBy row.Id
    select row
  }

  q |> Seq.iter (fun row -> printfn "%i %s" row.Id row.Name)

  Console.ReadLine() |> ignore
  0
```

Moim zdaniem SqlProvider jest najbardziej dojrzałym, przyjaznym i dającym największe możliwości dostawcą typów wśród dostawców bazodanowych. Oferuje dokładnie takie same możliwości jak SqlDataConnection, ale nie ogranicza się do jednej bazy. Dodatkowo, na etapie pisania kodu, umożliwia tworzenie zaawansowanych kwerend oraz daje dostęp do pojedynczych wierszy w tabeli.

TEXTFILETYPEPROVIDER

Po przybliżeniu przykładów wykorzystania najpopularniejszych dostawców przedstawię, w jaki sposób stworzyć własnego dostawcę, który będzie realizował nasz specyficzny scenariusz. Nasz dostawca będzie operował na plikach tekstowych. Dla podanej ścieżki do pliku tekstowego skonstruuje typ, który będzie udostępniał pewne właściwości tego pliku. Tworzenie własnego dostawcy rozpoczynamy od utworzenia projektu biblioteki F# (*class library*). Kolejnym krokiem jest podłączenie do projektu pakietu FSharp.TypeProviders.StarterPack, który pobierzemy jako paczkę nugetową. Po dołączeniu pakietu, do naszego projektu zostaną dodane pliki: *ProvidedTypes.fsi* oraz *ProvidedTypes.fs*. Pliki te dostarczają odpowiednich komponentów do utworzenia dostawcy. Nasz kod umieścimy w pliku *TextFileTypeProvider.fs*, w którym znajdzie się kod implementujący dostawcę. W Listingu 12 przedstawię implementację dostawcy, a następnie omówię najważniejsze elementy jego budowy.

Listing 12. Zawrtość pliku TextFileTypeProvider.fs

```
namespace MyTypeProviders.TextFileTypeProvider

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

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

  let namespaceName = "MyTypeProviders.TextFileTypeProvider"

  let thisAssembly = Assembly.GetExecutingAssembly()

  let staticParameters = [ProvidedStaticParameter("filePath",
  typeof<string>)]

  let providedTypeDefinition = ProvidedTypeDefinition(
    thisAssembly, namespaceName, "TextFile", Some typeof<obj>,
    HideObjectMethods = true)

  do providedTypeDefinition.DefineStaticParameters(
    parameters = staticParameters,
    instantiationFunction = (fun typeName paramValues ->
      match paramValues with
      | [| :? string as filePath |] ->
        let typeDefinition = ProvidedTypeDefinition(
          thisAssembly, namespaceName, typeName, Some typeof<obj>)

        let text = File.ReadAllText(filePath)

        let ctor = ProvidedConstructor(
          parameters = [], InvokeCode = fun args -> <@@ filePath
          :> obj @@>)
        ctor.AddXmlDoc "Initializes the TextFileTypeProvider instance."
        typeDefinition.AddMember ctor

        let textProperty = ProvidedProperty(
          "Text", typeof<string>, GetterCode = fun args -> <@@ text @@>)
        textProperty.AddXmlDoc "Gets the text from file."
        typeDefinition.AddMember textProperty

        let linesProperty = ProvidedProperty(
          "Lines", typeof<string[]>, GetterCode = fun args -> <@@
          text.Split('\n') @@>)
        linesProperty.AddXmlDoc "Gets all lines from file."
        typeDefinition.AddMember linesProperty

        let wordsProperty = ProvidedProperty(
          "Words", typeof<string[]>,
          GetterCode = fun args -> <@@ text.Split([|'\n'; ' ';
          ','; '.'; '?'; '!'|],
           StringSplitOptions.RemoveEmptyEntries) @@>)
        wordsProperty.AddXmlDoc "Gets all words from file."
        typeDefinition.AddMember wordsProperty

        typeDefinition
      | _ -> failwith "error"
    )
  )
  do this.AddNamespace(namespaceName, [providedTypeDefinition])

[<assembly:TypeProviderAssembly>]
do()
```

Zadaniem dostawcy jest utworzenie typu na podstawie przekazanych parametrów. W naszym przykładzie jest to ścieżka do pliku tekstowego. Dostarczając typ, musimy stworzyć jego strukturę i umieścić go w przestrzeni nazw. Nasz dostawca udostępnia następujące propercje: `Text` – cały tekst z pliku, `Lines` – udostępnia kolekcję linii z pliku oraz `Words` – wszystkie słowa z pliku. Propercje tworzymy z pomocą metody `ProvidedProperty`, którą dodajemy do definicji typu – `typeDefinition.AddMember`. Dodatkowo za pomocą `AddXmlDoc` dodajemy komentarz do propercji, który będzie widoczny podczas korzystania z IntelliSense. Do przetestowania dostawcy użyjemy skryptu F# (pliku *test.fsx*). Jako dane wejściowe posłuży plik tekstowy z losowym tekstem, popularnym „Lorem ipsum".

Listing 13. Zawartość pliku test.fsx

```
#r @".\bin\TextFileTypeProvider.dll"

type loremIpsumType = MyTypeProviders.TextFileTypeProvider.
TextFile<"C:\LoremIpsum.txt">

let loremIpsum = new loremIpsumType()

"Text from file:" |> System.Console.WriteLine
loremIpsum.Text |> System.Console.WriteLine
"" |> System.Console.WriteLine

"Lines count:" |> System.Console.WriteLine
loremIpsum.Lines.Length |> System.Console.WriteLine
"" |> System.Console.WriteLine

"First line:" |> System.Console.WriteLine
loremIpsum.Lines.[0] |> System.Console.WriteLine
"" |> System.Console.WriteLine

"Words count:" |> System.Console.WriteLine
loremIpsum.Words.Length |> System.Console.WriteLine
"" |> System.Console.WriteLine

"First and second words:" |> System.Console.WriteLine
loremIpsum.Words.[0] + " " + loremIpsum.Words.[1] |> System.
Console.WriteLine
"" |> System.Console.WriteLine

// Text from file:
// Lorem ipsum dolor sit amet...

// Lines count:
// 3

// First line:
// Lorem ipsum dolor sit amet, consectetur adipiscing elit.

// Words count:
// 55

// First and second words:
// Lorem ipsum
```

Dostawcy typów to narzędzia dające spore możliwości i zwiększające produktywność. Ułatwiają pracę z danymi z różnych źródeł poprzez zautomatyzowany proces tworzenia typów mapujących ich strukturę. Mając typy, od razu możemy sięgać po dane, co znacznie skraca tworzony kod.

PAWEŁ WICHER
pawel.wicher@op.pl

Programista .NET. Od kilku lat zajmuje się komercyjnym tworzeniem aplikacji w oparciu o stos technologiczny Microsoft. Aktywny na codewars. W wolnych chwilach gra w piłkę nożną, uczy się nowych języków programowania oraz gra na basie.