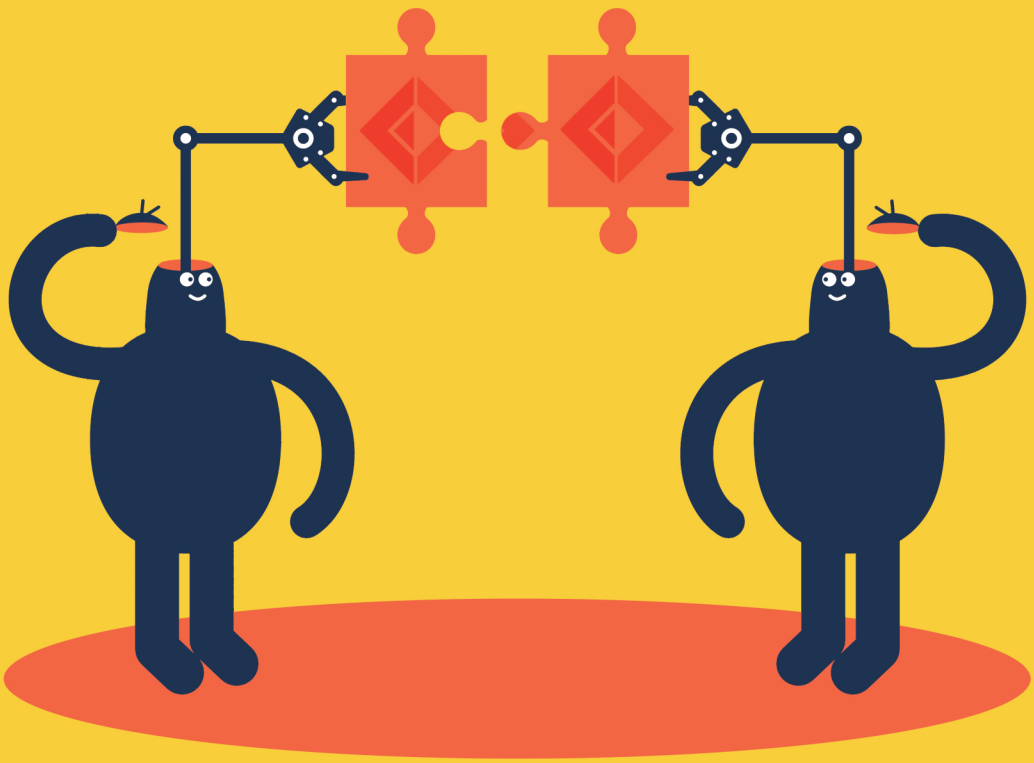
GET PROGRAMMING WITH F#

A guide for .NET developers



Isaac Abraham

Forewords
by Dustin Campbell
and Tomas Petricek

MANNING

SAMPLE
CHAPTER

Get Programming with F#
by Isaac Abraham

Lesson 30

Contents

Foreword **v**
Preface **viii**
Acknowledgments **x**
About this book **xi**
About the author **xiv**

Welcome to Get Programming with F#! **1**

Unit 1

F# AND VISUAL STUDIO

Lesson 1 The Visual Studio experience **17**
Lesson 2 Creating your first F# program **25**
Lesson 3 The REPL—changing how we develop **34**

Unit 2

HELLO F#

Lesson 4 Saying a little, doing a lot **47**
Lesson 5 Trusting the compiler **58**
Lesson 6 Working with immutable data **70**
Lesson 7 Expressions and statements **81**
Lesson 8 Capstone 1 **92**

Unit 3

TYPES AND FUNCTIONS

Lesson 9 Shaping data with tuples **101**
Lesson 10 Shaping data with records **111**
Lesson 11 Building composable functions **125**
Lesson 12 Organizing code without classes **138**
Lesson 13 Achieving code reuse in F# **149**
Lesson 14 Capstone 2 **160**

Unit 4

COLLECTIONS IN F#

Lesson 15 Working with collections in F# **173**
Lesson 16 Useful collection functions **186**
Lesson 17 Maps, dictionaries, and sets **197**
Lesson 18 Folding your way to success **206**
Lesson 19 Capstone 3 **219**

Unit 5

THE PIT OF SUCCESS WITH THE F# TYPE SYSTEM

Lesson 20 Program flow in F# **231**
Lesson 21 Modeling relationships in F# **244**
Lesson 22 Fixing the billion-dollar mistake **257**
Lesson 23 Business rules as code **270**
Lesson 24 Capstone 4 **284**

Unit 6

LIVING ON THE .NET PLATFORM

Lesson 25 Consuming C# from F# **299**
Lesson 26 Working with NuGet packages **310**

Lesson 27 Exposing F# types and functions to C# **321**
Lesson 28 Architecting hybrid language applications **331**
Lesson 29 Capstone 5 **342**

Unit 7

WORKING WITH DATA

Lesson 30 Introducing type providers **355**
Lesson 31 Building schemas from live data **365**
Lesson 32 Working with SQL **376**
Lesson 33 Creating type provider-backed APIs **388**
Lesson 34 Using type providers in the real world **401**
Lesson 35 Capstone 6 **411**

Unit 8

WEB PROGRAMMING

Lesson 36 Asynchronous workflows **425**
Lesson 37 Exposing data over HTTP **439**
Lesson 38 Consuming HTTP data **453**
Lesson 39 Capstone 7 **464**

Unit 9

UNIT TESTING

Lesson 40 Unit testing in F# **477**
Lesson 41 Property-based testing in F# **489**
Lesson 42 Web testing **501**
Lesson 43 Capstone 8 **511**

Unit 10

WHERE NEXT?

Appendix A The F# community **521**
Appendix B F# in my organization **527**
Appendix C Must-visit F# resources **537**
Appendix D Must-have F# libraries **543**
Appendix E Other F# language features **556**

Index **564**

30 LESSON

INTRODUCING TYPE PROVIDERS

Welcome to the world of data! The first lesson of this unit will

- Gently introduce you to type providers
- Get you up to speed with the most popular type provider, FSharp.Data

After this lesson, you'll be able to work with external data sources in various formats more quickly and easily than you've ever done before in .NET—guaranteed!

30.1 Understanding type providers

Type providers are a language feature first introduced in F# 3.0:

> *An F# type provider is a component that provides types, properties, and methods for use in your program. Type providers are a significant part of F# 3.0 support for information-rich programming.*
>
> —https://docs.microsoft.com/en-us/dotnet/articles/fsharp/tutorials/type-providers/index

At first glance, this sounds a bit fluffy. You already know what types, properties, and methods are. And what does *information-rich programming* mean? The short answer is to think of type providers as T4 templates on steroids—a form of code generation, but one that lives *inside* the F# compiler. Confused? Read on.

Let's look at a somewhat holistic view of type providers first, before diving in and working with one to see what the fuss is all about. You might already be familiar with the notion of a compiler that parses C# (or F#) code and builds IL from which you can run applications, and if you've ever used Entity Framework (particularly the earlier versions) or old-school SOAP web services in Visual Studio, you're familiar with the idea of code generation tools such as T4 templates. These are tools that can generate C# code from another language or data source, as depicted in figure 30.1.

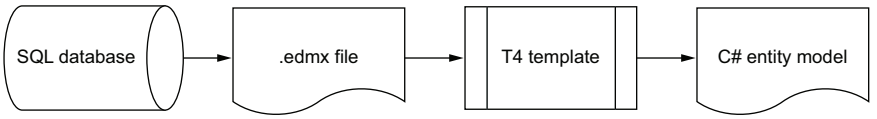


Figure 30.1 Entity Framework database-first code generation process

In this example, Entity Framework has a tool that can read a SQL database schema and generate an .edmx file—an XML representation of a database. From here, a T4 template is used to generate C# classes that map back to the SQL database.

Ultimately, T4 templates and the like, although useful, are awkward to use. For example, you need to attach them into the build system to get them up and running, and they use a custom markup language with C# embedded in them; they're not great to work with or distribute.

At their most basic, type providers are just F# assemblies (which anyone can write) that can be plugged into the F# compiler, and can then be used at edit time to generate entire type systems for you to work with *as you type*. In a sense, type providers serve a similar purpose to T4 templates, except they're much more powerful, more extensible, more lightweight to use, and extremely flexible. They can be used with what I call *live* data sources, and also offer a gateway not just to data sources but also to other *programming languages*, as shown in figure 30.2.

Unlike T4 templates, type providers can affect type systems without rebuilding the project, because they run in the background *as you write code*. Dozens, if not hundreds, of type providers are available, from ones that work with simple flat files, to relational SQL databases, to cloud-based data storage repositories such as Microsoft Azure Storage or Amazon Web Services S3. The term *information-rich* programming refers to the concept of bringing disparate data sources into the F# programming language in an extensible way.

Don't worry if that sounds a little confusing. You'll take a look at your first type provider in just a second.

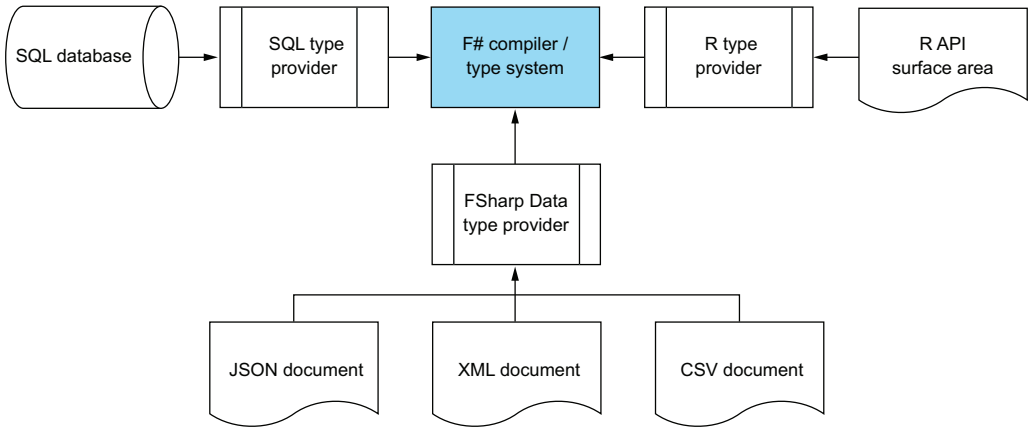


Figure 30.2 A set of F# type providers with supported data sources

Quick check 30.1

1. What is a type provider?
2. How do type providers differ from T4 templates?
3. Is the number of type providers fixed?

30.2 Working with your first type provider

Let's look at a simple example of a data access challenge, not unlike what you looked at in lesson 13. You'll work with soccer results, except that rather than working with an in-memory dataset, you'll work with a larger, external data source—a CSV file, located in learnfsharp/data/ FootballResults.csv. You need to answer the following question: which three teams won at home the most over the whole season?

QC 30.1 answer

1. A flexible code generation mechanism supported by the F# compiler.
2. Type providers are supported within the F# compiler directly, and allow edit-time type generation; there's no code generation as with T4 templates.
3. No. Type providers can be written, downloaded, and added to your applications as separate, reusable components.

30.2.1 Working with CSV files today

Let's first think about the typical process that you might use to answer this question, as shown in figure 30.3.

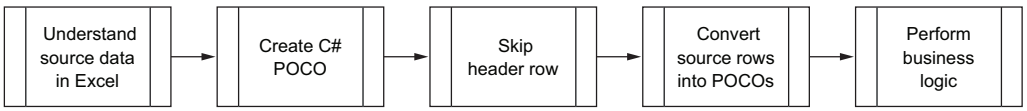


Figure 30.3 Steps to parse a CSV file in order to perform a calculation on it

Before you can even begin to perform the calculation, you need to *understand* the data. This usually means looking at the source CSV file in Excel or a similar program, and then designing a C# type to match the data in the CSV. Then, you do all of the usual boilerplate parsing: opening a handle to the file, skipping the header row, splitting on commas, pulling out the correct columns, and parsing into the correct data types. Only after doing all of that can you start to work with the data and produce some business value. Most likely, you'll use a console application to get the results, too. This entire process is more akin to typical software engineering—not a great fit when you want to explore data quickly and easily.

30.2.2 Introducing FSharp.Data

You could quite happily perform the preceding steps in F#; at least using the REPL affords you a more exploratory way of development. But that process wouldn't remove the whole boilerplate element of parsing the file, and this is where your first type provider comes in: FSharp.Data.

FSharp.Data is an open source, freely distributable NuGet package designed to provide generated types when working with data in CSV, JSON, or XML formats.

Using scripts for the win

At this point, I'm going to advise you to move away from heavyweight solutions and start to work exclusively with standalone scripts; this fits much better with what you're going to be doing. You'll notice in the code repository a build.cmd file—run it. This command uses Paket to download NuGet packages into the packages folder, which you can then reference directly in your scripts. This means you don't need a project or solution to start coding—you can simply create scripts and jump right in. I recommend creating your scripts in the src/code-listings/ folder (or another folder at the same level,

such as src/learning/) so that the package references shown in the listings here work without needing changes.

Working with CSV files

Let's look at our first experiment with a type provider, the CSV type provider in FSharp.Data, and perform the analysis that we discussed at the start of this section.

Now you try

You'll start by doing some simple data analysis over a CSV file:

1. Create a new standalone script in Visual Studio by choosing File > New. You don't need a solution here; remember that a script can work standalone.
2. Save the newly created file into an appropriate location as described in "Scripts for the win."
3. Enter the following code.

Listing 30.1 Working with CSV files using FSharp.Data

Referencing the FSharp.Data assembly

Connecting to the CSV file to provide types based on the supplied file

```
#r @"..\..\packages\FSharp.Data\lib\net40\FSharp.Data.dll"
open FSharp.Data
type Football = CsvProvider< @"..\..\data\FootballResults.csv">
let data = Football.GetSample().Rows |> Seq.toArray
```

Loading in all data from the supplied CSV file

That's it. You've now parsed the data, converted it into a type that you can consume from F#, and loaded it into memory. Don't believe me? Check out figure 30.4.

Figure 30.4 Accessing a provided type from FSharp.Data

You now have full IntelliSense to the dataset. That's it! You don't have to manually parse the dataset; that's been done for you. You also don't need to figure out the types; the type provider will scan through the first few rows and infer the types based on the contents of the file! Rather than using a tool such as Excel to understand the data, you can now begin to use F# as a tool to both understand *and* explore your data.

Backtick members

You'll see in figure 30.4, as well as from the code when you try it out yourself, that the fields listed have spaces in them! It turns out that this isn't a type provider feature, but one that's available throughout F# called *backtick members*. Just place a double backtick (``) at the beginning and end of the member definition, and you can put spaces, numbers, or other characters in the member definition. Visual Studio doesn't correctly provide IntelliSense for these in all cases (for example, `let`-bound members on modules), but it works fine on classes and records. You'll see some interesting uses for this when dealing with unit testing.

Visualizing data

While we're at it, let's also look at an easy-to-use F#-friendly charting library, XPlot. This library provides access to charts available in Google Charts as well as Plotly. You'll use the Google Charts API here, which means adding dependencies to XPlot.GoogleCharts (which also brings down the Google.DataTable.Net.Wrapper package):

1. Add references to both the XPlot.GoogleCharts and Google.DataTable.Net.Wrapper assemblies. If you're using standalone scripts, both packages will be in the packages folder after running build.cmd. Use `#r` to reference the assembly inside one of the lib/net folders.
2. Open the XPlot.GoogleCharts namespace.
3. Execute the following code to calculate the results and plot them as a chart, as shown in figure 30.5.

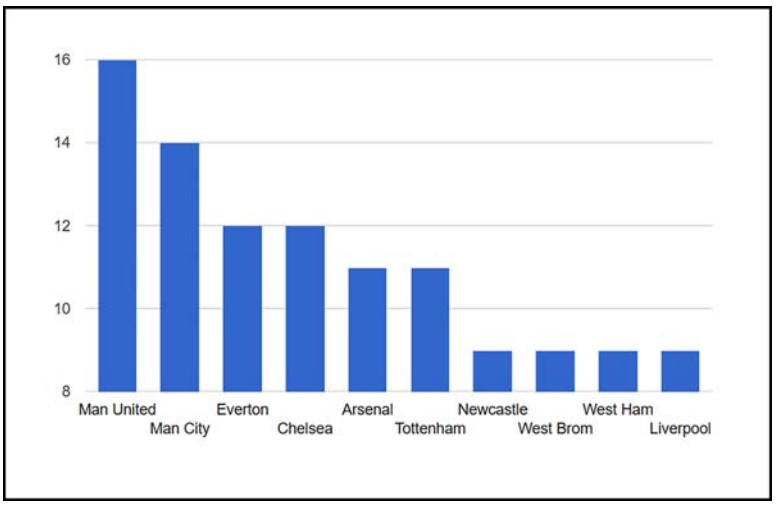


Figure 30.5 Visualizing data sourced from the CSV type provider

Listing 30.2 Charting the top ten teams for home wins

```
data
|> Seq.filter(fun row ->
    row.``Full Time Home Goals`` > row.``Full Time Away Goals``)
|> Seq.countBy(fun row -> row.``Home Team``)
|> Seq.sortByDescending snd
|> Seq.take 10
|> Chart.Column
|> Chart.Show
```

countBy generates a sequence of tuples (team vs. number of wins).

Converting the sequence of tuples into an XPlot Column Chart

Showing the chart in a browser window

In a few lines of code, you were able to open a CSV file you've never seen, explore the schema of it, perform an operation on it, and then chart it in less than 20 lines of code—not bad! This ability to rapidly work with and explore datasets that you haven't even seen before, while still allowing you to interact with the full breadth of .NET libraries that are out there, gives F# unparalleled abilities for bringing in disparate data sources to full-blown applications.

Type erasure

The vast majority of type providers fall into the category of *erasing* type providers. The upshot of this is that the types generated by the provider exist only at *compile* time. At runtime, the types are erased and usually compile down to plain objects; if you try to use reflection over them, you won't see the fields that you get in the code editor.

One of the downsides is that this makes them extremely difficult (if not impossible) to work with in C#. On the flip side, they're extremely efficient. You can use erasing type providers to create type systems with thousands of types without any runtime overhead, because at runtime they're of type `Object`.

Generative type providers allow for runtime reflection, but are much less commonly used (and from what I understand, much harder to develop).

30.2.3 Inferring types and schemas

One of the biggest differences in terms of mind-set when working with type providers is the realization that the type system is driven by an external data source. This schema may be inferred, as you saw with the CSV provider. Let's see a quick example of how this can affect your development process:

1. In your script, change the data source for the CSVProvider from FootballResults.csv to FootballResultsBad.csv. This version of the CSV file has had the contents of the Away Goals column changed from numbers to strings.
2. You'll immediately notice a compile-time error within your query, as shown in figure 30.6.

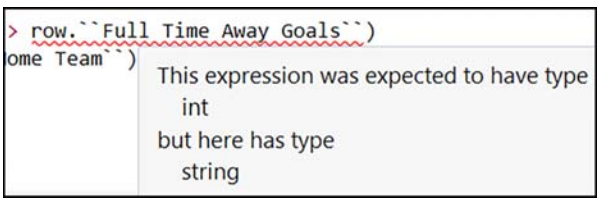


Figure 30.6 Changes in inferred schema cause compile-time errors.

This is because the type provider has inferred the types based on the contents of the sheet.

This point is crucial to grasp, within the context of not only a script, but also a full-blown application. Imagine you're compiling your application off a CSV file provided by your customer, and one day that customer provides you with a new version of the format. You can supply the new file to your code and instantly know where incompatibles in your code are; any breaking changes won't compile. This sort of instant feedback is much quicker than either unit tests or runtime errors. Instead, you're using the compiler and type system—the earliest possible stage—to show you exactly where code breaks. Also, in case you're wondering, type providers support the ability to redirect from one file to another so that you can compile against one but run against another. You'll deal with this in the coming lessons.

Finally, note that when it comes to schema inference, some type providers work differently from others. For example, FSharp.Data allows you to manually override the schema by supplying a custom argument to the type provider. Others can use some form of schema guidance from the source system. For example, SQL Server provides rich schema information from which a type provider doesn't need to infer types at all.

Writing your own type providers?

Sorry—but this book doesn't cover how to write your own type providers! The truth is that they're not easy to develop—particularly testing them while you develop them—but a few decent resources are worth looking at, such as the Starter Pack (https://github.com/fsprojects/FSharp.TypeProviders.StarterPack), as well as online video courses. If you're interested in learning how to write your own, I strongly advise you to look at some of the simpler ones to start with, or try to contribute to one of the many open source type providers; this is probably the best way to learn how they work.

Quick check 30.2

1. What are erased types?
2. What are backtick members?

QC 30.2 answer

1. Erased types are types that exist at compile-time only; at runtime, they're "erased" to objects.
2. Backtick members are members of a type surrounded with double backticks, which allow you to enter spaces and characters that would normally be forbidden in the name.

Summary

In this lesson, you took your first look at type providers. The remainder of this unit will introduce you to other forms of type providers and give you an idea of how far they can go. In this lesson

- You saw what type providers are at a high level and learned about some of their uses.
- You explored the FSharp.Data package and saw how to work with CSV files.
- You saw the XPlot library, a charting package that's designed to work well with F#.

Try this

Find any CSV file that you have on your PC. Try to parse it by using the CSV type provider and perform simple operations on it, such as list aggregation. Or download a CSV containing data and from the internet and try with that!

Alternatively, try creating a more complex query to compare the top five teams that scored the most goals and display it in a pie chart.

SOFTWARE DEVELOPMENT

GET PROGRAMMING WITH F#

A guide for .NET developers | Isaac Abraham

Your .NET applications need to be good for the long haul. F#'s unique blend of functional and imperative programming is perfect for writing code that performs flawlessly now and keeps running as your needs grow and change. It takes a little practice to master F#'s functional-first style, so you may as well get programming!

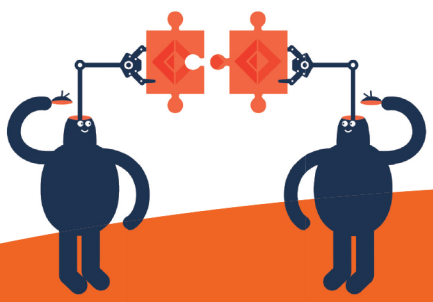
Get Programming with F#: A guide for .NET developers teaches F# through 43 example-based lessons with built-in exercises so you can learn the only way that really works: by practicing. The book upgrades your .NET skills with a touch of functional programming in F#. You'll pick up core FP principles and learn techniques for iron-clad reliability and crystal clarity. You'll discover productivity techniques for coding F# in Visual Studio, functional design, and integrating functional and OO code.

WHAT'S INSIDE

- **Learn how to write bug-free programs**
- **Turn tedious common tasks into quick and easy ones**
- **Use minimal code to work with JSON, CSV, XML, and HTML data**
- **Integrate F# with your existing C# and VB.NET applications**
- **Create web-enabled applications**

Written for intermediate C# and Visual Basic .NET developers.
No experience with F# is assumed.

Isaac Abraham *is an experienced .NET developer and trainer. He's an F# MVP for his contributions to the .NET community.*



To download their free eBook *in PDF, ePub, and Kindle formats, owners of this book should visit*
manning.com/books/get-programming-with-f-sharp

MANNING

US $44.99 | Can $59.99 [Including eBOOK]

FREE EBOOK
See first page

"Leads you on a journey of F# that's both pragmatic and relevant."
– FROM THE FOREWORD BY DUSTIN CAMPBELL MICROSOFT

"Does a great job of explaining F# clearly and effectively."
– FROM THE FOREWORD BY TOMAS PETRICEK FSHARPWORKS

"A wonderful introduction to the subtleties of the F# language. You'll be productive within minutes!"
– JASON HALES DIGITAL TIER

"Combines excellent explanations, real-world use cases, and a steady supply of questions and exercises to make sure you really understand what is being taught. Highly recommended!"
– JOEL CLERMONT GROWTHPOINT

"Puts the FUN into functional programming with F#."
– STEPHEN BYRNE ACTION POINT

ISBN-13: 978-1-61729-399-3
ISBN-10: 1-61729-399-7
54499
9 781617 293993