CHAPTER 13

Integrating External Data and Services

One of the major trends in modern computing is "the data deluge," meaning the rapid rise in availability of massive quantities of digital information available for analysis, especially through reliable networked services and large-scale data storage systems. Some of this data may be collected by an individual or an organization, some may be acquired from data providers, and some from free data sources.

In this world, programmed services and applications can be viewed as components that consume, filter, transform and re-republish information within a larger connected and reactive system. Nearly all modern software components or applications incorporate one or more external information sources. This may include static data in tables, static files copied into your application, data coming from relational databases, data coming from networked services (including the web or the local enterprise network), or data from the ambient context of sensors and devices. When the whole web is included, there are an overwhelming number of data sources which you might use, and hundreds more appearing every month! The digital world is exploding with information, almost beyond imagination.

For example, consider the data provided by the World Bank (`http://api.worldbank.org`), which includes thousands of data sets for hundreds of countries. From one perspective, this is a lot of data: it would take you a long time to learn and explore all this, and there are a lot of practical things you can do with it. However, from another perspective, the World Bank data is *tiny*: it is just one data source hidden in one remote corner of the Internet which you've probably never heard of until just now.

Because the world of external data is so big, in this chapter we can't hope to describe how to use every data source, or even every "kind" of data source. We also can't cover the incredible range of things you might want to do with different data sources. Instead, this chapter introduces a selection of topics in data-rich programming. Along the way you get to use and learn some important and innovative F# features related to this space.

- You first learn some simple and pragmatic techniques to work with external web data using HTTP REST requests, XML, and JSON.
- You then look at how F# 3.0 allows for language integration of some schematized data sources through the *type provider* feature introduced in this version of the language. You learn how to use the F# built-in type providers to query OData REST services and SQL databases directly from F# code in a more immediate way.
- You then learn how to author more advanced queries, including sorting, grouping, joins, and statistical aggregates.

D. Syme et al., *Expert F# 3.0*

- You then look at how to use the lower-level ADO.NET libraries for some data programming tasks for relational databases.

Along the way, remember that the role of F# is as a workhorse to help control the complexity of working with external data.

Some Basic REST Requests

We begin our adventures in working with external information sources by looking at some simple examples of making HTTP REST requests to web-hosted resources. You have already used the basic code to make an HTTP request in Chapters 2 and 3:

```
open System.IO
open System.Net

let http (url: string) =
    let req = WebRequest.Create(url)
    use resp = req.GetResponse()
    use stream = resp.GetResponseStream()
    let reader = new StreamReader(stream)
    reader.ReadToEnd()
```

You have used this function to fetch web pages containing HTML, but it can also be used to fetch structured data in formats such as XML and JSON from web-hosted services. For example, one such service is the World Bank API mentioned in the introduction to this chapter. This service exposes a data space containing *regions*, *countries* and *indicators*, where the indicators are a time-series of time/value pairs. You can fetch the list of countries by using:

```
let worldBankCountriesXmlPage1 = http "http://api.worldbank.org/country"
```

This fetches the first page of the country list as XML, and the results will be like this:

```
val worldBankCountriesXmlPage1 : string =
  "<?xml version="1.0" encoding="utf-8"?>
<wb:countries page="1"+[25149 chars]

> worldBankCountriesXmlPage1;;

val it : string =
  "<?xml version="1.0" encoding="utf-8"?>
<wb:countries page="1" pages="5" per_page="50" total="246" xmlns:wb="http://www.worldbank.
org">
  <wb:country id="ABW">
    <wb:iso2Code>AW</wb:iso2Code>
    <wb:name>Aruba</wb:name>
    <wb:region id="LCN">Latin America &amp;amp; Caribbean (all income levels)</wb:region>
    <wb:adminregion id="" />
    <wb:incomeLevel id="NOC">High income: nonOECD</wb:incomeLevel>
    <wb:lendingType id="LNX">Not classified</wb:lendingType>
    <wb:capitalCity>Oranjestad</wb:capitalCity>
    <wb:longitude>-70.0167</wb:longitude>
    <wb:latitude>12.5167</wb:latitude>
```

```
  </wb:country>
  <wb:country id="AFG">
...
  </wb:country>
...
</wb:countries>"
```

Getting Data in JSON Format

Web data sources typically support both XML and JSON result formats. For example, you can fetch the same data in JSON format as follows:

```
let worldBankCountriesJsonPage1 = http "http://api.worldbank.org/country?format=json"
```

The results will then look like this:

```
val worldBankCountriesJsonPage1 : string =
  "[{"page":1,"pages":5,"per_page":"50","total":246},[{"id":"ABW"+[17322 chars]
```

Parsing the XML or JSON Data

Once we have the resulting XML or JSON data back from the service, we can parse it using techniques you have already seen from Chapters 8 and 9:

```
#r "System.Xml.Linq.dll"
open System.Xml.Linq

/// The schema tag for this particular blob of XML
let xmlSchemaUrl = "http://www.worldbank.org"

// General utilities for XML
let xattr s (el: XElement) = el.Attribute(XName.Get(s)).Value
let xelem s (el: XContainer) = el.Element(XName.Get(s, xmlSchemaUrl))
let xelems s (el: XContainer) = el.Elements(XName.Get(s, xmlSchemaUrl))
let xvalue (el: XElement) = el.Value

/// The results of parsing
type Country = { Id: string; Name: string; Region: string; }

/// Parse a page of countries from the WorldBank data source
let parseCountries xml =
    let doc = XDocument.Parse xml
    [ for countryXml in doc |> xelem "countries" |> xelems "country" do
        let region = countryXml |> xelem "region" |> xvalue
        yield
            {
                Id = countryXml |> xattr "id"
                Name = countryXml |> xelem "name" |> xvalue
                Region = region
            }]
```

This extracts some of the fields from each of the <country> nodes in the returned XML. When we execute the XML parsing over the first page of input data, we get:

```
> parseCountries worldBankCountriesXmlPage1;;
val it : Country list =
  [{Id = "ABW";
    Name = "Aruba";
    Region = "Latin America & Caribbean (all income levels)";};
   ...
   {Id = "CYM";
    Name = "Cayman Islands";
    Region = "Latin America & Caribbean (all income levels)";}]
```

Parsing data in JSON format can be done in a similar way using a library such as NewtonSoft's Json.NET, from json.codeplex.com.

Handling Multiple Pages

Web-hosted services like the WorldBank API are "stateless" – they will essentially respond to identical requests with identical responses. Accessing services like this uses a technique known as REST, which stands for *REpresentational State Transfer*. This means that any "state" is passed back and forth between the client and the services as data.

One simple example of this works is how we collect multiple pages of data from the services. So far, we have only retrieved the first page of countries. If we want to collect all the pages, we need to make multiple requests, where we give a page parameter. For example, we can get the second page as follows:

```
let worldBankCountriesXmlPage2 = http "http://api.worldbank.org/country?page=2"
```

Note that the "state" of the process of iterating through the collection is passed simply by using a different URL, one with "page=2" added. This is very typical of REST requests.

The total number of pages can be fetched by parsing the first page, which contains the total page count as an attribute. If using XML, we use the following:

```
let doc = XDocument.Parse worldBankCountriesXmlPage1
let totalPageCount = doc |> xelem "countries" |> xattr "pages" |> int
```

When executed, this reveals a total page count of 5:

```
val totalPageCount : int = 5
```

We can now put this together into a function that fetches all the pages through a sequence of requests, as follows:

```
let rec getCountryPages() =
    let page1 = http "http://api.worldbank.org/country"
    let doc1 = XDocument.Parse page1
    let numPages = doc1 |> xelem "countries" |> xattr "pages" |> int
    let otherPages =
        [ for i in 2 .. numPages ->
              http ("http://api.worldbank.org/country?page=" + string i)]
    [ yield! parseCountries page1
```

```
    for otherPage in otherPages do
        yield! parseCountries otherPage ]
```

When executed, this reveals a total country count of 246 at the time of writing:

```
> getCountryPages() |> Seq.length;;

val it : int = 246
```

As an alternative, you can often simply increase a REST parameter that indicates how many results to return per page. For example, in the case of the WorldBank API this is the `per_page` parameter, e.g. this URL:

```
http://api.worldbank.org/country?format=json&per_page=1000
```

This will return all the results in a single page which can then be parsed. However, not all services allow arbitrarily high numbers of items per page so parsing multiple pages is still often necessary.

Getting Started with Type Providers and Queries

In the first section you performed some basic REST requests in F#. However, you will have noticed that working with external information in this way is somewhat tedious because you must manually construct REST API calls and manually parse the XML or JSON returned by the service. This gets progressively harder as services become more complex, as services change, and as additional features like authentication are required. Further, it is very easy to make mistakes in all parts of this process, including parsing the XML or JSON.

In this section, we look at a unique F# language/tooling feature called *type providers* which is designed to make information sources directly available in the F# language in simpler, more intuitive, and more directly strongly-typed way.

Example - Language Integrated OData

Our first example is accessing an internet data protocol using F# programming. We use "OData" (see `www.odata.org`) as our example. Here is the code to access the data service:

```
#r "System.Data.Services.Client.dll"
#r "FSharp.Data.TypeProviders.dll"
open Microsoft.FSharp.Data.TypeProviders

type Northwind =
    ODataService<"http://services.odata.org/Northwind/Northwind.svc/">

let db = Northwind.GetDataContext()
```

In this code, you first reference two libraries: the "runtime" library for OData services `System.Data.Services.Client`, and the F# library for the F# 3.0 OData type provider `FSharp.Data.TypeProviders.dll`. You can now get data from the service as follows:

```
let first10Customers =
    query { for c in db.Customers do
            take 10
```

```
            select c }
    |> Seq.toList
```

When run, this gives the first 10 customers from the OData service:

```
> first10Customers;;
val it : Northwind.ServiceTypes.Customer list =
  [Customer {Address = "Obere Str. 57";
             City = "Berlin";
             CompanyName = "Alfreds Futterkiste";
             ContactName = "Maria Anders";
             ContactTitle = "Sales Representative";
             Country = "Germany";
             CustomerDemographics = seq [];
             CustomerID = "ALFKI";
             Fax = "030-0076545";
             Orders = seq [];
             Phone = "030-0074321";
             PostalCode = "12209";
             Region = null;};...]
```

To understand what's going on, it's helpful to add the following line before executing the code above:

```
service.DataContext.SendingRequest.Add (fun x -> printfn "requesting %A" x.Request.RequestUri)
```

After you add this, you will see that the output begins:

```
requesting http://services.odata.org/Northwind/Northwind.svc/Customers()?$top=10
```

As you can see, OData is a protocol for querying data sources over HTTP, and is ultimately implemented by REST requests. The code above accesses the data service at `services.odata.org/Northwind/Northwind.svc`. Under the hood, an OData service request is a URL with embedded strings to represent queries. The response text is JSON or XML, which is automatically turned back into objects for you.

What is a Type Provider?

The example above uses two things that you have not yet seen in this book: a use of a type provider, and a use of an F# query. The use of the type provider is here:

```
open Microsoft.FSharp.Data.TypeProviders

type Northwind =
    ODataService<"http://services.odata.org/Northwind/Northwind.svc/">
```

A type provider is a compile-time component that, given optional static parameters identifying an external information space, provides two things to the host F# compiler/tooling:

- A component "signature" that acts as the programming interface to that information space, and which is computed on-demand as needed by the F# compiler. For F#, the component signature contains provided namespaces, types, methods, properties,

events, attributes, and literals that give a .NET object-oriented characterization of the information space.

- An implementation of the component signature. This is given by either an actual .NET assembly that implements the component signature (the generative model for the provided types), or a pair of erasure functions giving representation types and representation expressions for the provided types and provided methods respectively (the erasure model for the provided types).

Put simply, type providers are about using a provider model for the "type import" logic of the host language compiler or tooling. Essentially, a type provider is an adapter component that reads schematized data and services and transforms them into types in the target programming language. This allows programmers to quickly leverage rich, schematized information sources without an explicit transcription process (be it code generation or a manually created ontology). The provided types can then be leveraged by not only the type-checker and runtime, but also by tools that rely on the type-checker, such as IDE auto-completion. Also, if the data source contains additional descriptive metadata (such as a description of various columns in a database), this can be transformed by the type provider into information that is visible to the programmer within the IDE (such as documentation contained in tooltips).

A type provider does not necessarily contain any types itself; rather, it is a component for generating descriptions of types, methods, and their implementations. A type provider is thus a form of compile-time meta-programming: a compiler plugin that augments the set of types that are known to the type-checker and compiler. Importantly, a type provider can provide types and methods on-demand, i.e. lazily, as the information is required by the host tool such as the F# compiler. This allows the provided type space to be very large or even infinite.

What is a Query?

The second new construct in the code above is a *query*, in particular this program text:

```
query { for c in db.Customers do
        take 10
        select c }
```

If you know C# you will recognize this as a way of writing LINQ queries. A LINQ query is simply a way of specifying a query of a data source using a set of operators such as `select`, `where`, `take`, and so on. In F#, the code inside `query { … }` is converted by a "LINQ query provider" into an actual request that is sent to the data provider–in this particular case, a REST request sent to the OData service. This translation relies on the F# compiler to insert quotations automatically, and using the underlying quotation representation to convert the inner F# expression to a LINQ query that can be iterated over. You've seen one example already of how this translation works, where the code above becomes:

```
requesting http://services.odata.org/Northwind/Northwind.svc/Customers()?$top=10
```

You can experiment with this particular implementation of query translation from F# queries to OData. For example, the simpler:

```
query { for c in db.Customers do
        select c }
```

uses the URL:

```
requesting http://services.odata.org/Northwind/Northwind.svc/Customers()
```

and this more complex query:

```
query { for c in db.Customers do
        where (c.ContactName.Contains "Maria")
        take 10
        select c }
```

yields this request:

```
requesting http://services.odata.org/Northwind/Northwind.svc/Customers()
?$filter=substringof('Maria',ContactName)&$top=10
```

You will learn more about queries and query translation later in this chapter.

QUERYING ODATA – NOT QUITE AS EASY AS IT LOOKS

This section has used the F# 3.0 support for the OData protocol as an example of integrating a remote data source into F# programming. While OData gives a great "in-the-box" example of F# type providers, it turns out that querying OData is actually quite a bit harder than we've made it look in this section. This isn't really anything to do with F#—it is mostly to do with the OData query implementation itself in the .NET Framework and because so far we've ignored some important issues like authentication.

So, in the name of full disclosure, we state openly that querying OData with F# 3.0 (or C# 5.0) is actually full of a surprising number of subtleties. Here is a quick guide:

- **Queries should select an entity**. Select an entity (e.g., a customer `c`) rather than a property of an entity (such as `c.Name`). Selecting a property of an entity gives a cryptic error message about "Navigation properties."
- **Queries returning simple entities are easy**. Simple entities have properties which are all primitive types like integers, strings, dates, and GUIDs.
- **Queries returning complex entities need "Expand."** If the data being returned contains further entities (e.g. a `Customer` contains a list of `Orders`), then things become surprisingly harder and you must "expand" those entities. If not, the corresponding data will appear as empty! You will see examples of how to do this later in this chapter. A quick example is:

  ```
  query { for c in db.Customers.Expand("Orders") do
          select c }
  ```

- **If you expand, only query the DataContext once**. When you expand some nested entities, these get placed in the reified graph of objects for the OData context. If you have already queried the data context, you will not see the expanded data. *So, if you need to expand, make sure you haven't already used the data context for other queries against that collection!*
- **Learn what you can use in filters.** Filters (`where`) can use a range of methods. For example, on strings you can use `.StartsWith`, `.EndsWith`, `.Replace`, `.ToLower`, `.ToUpper`, `.Trim`, `.Substring`, `.Length` and on `DateTime` values you can use `.Year`, `.Month`, `.Day`, `.Hour`, `.Minute`, `.Second`.

- * **Learn what you can use in queries.** Queries can use `from`, `where`, `sortBy`, `thenBy`, `select`, `skip` and `take` plus uses of `.Expand`. You can't really use anything else in OData queries. Don't try.
- **Learn how to get paged data.** OData service implementations usually only return the first 20 to 100 results of a query. To get the remaining results you need to use "pagination." This is a common pattern for all service-oriented programming. You will see examples of doing OData pagination later in this chapter.
- **If using credentials, set them in the SendingRequest callback.** OData services may need credentials, usually either using Basic or OAuth authentication. To set them, use the following for Basic:

```
db.DataContext.SendingRequest.Add(fun e ->
    e.RequestHeaders.["Authorization"] <- "Basic " + base64 encoding of
    "username:password")
```

and the following for OAuth:

```
db.DataContext.SendingRequest.Add(fun e ->
    e.RequestHeaders.["Authorization"] <- "OAuth " + securityToken)
```

- **On some platforms, you need to use asynchronous requests.** In Silverlight and Windows 8, the OData implementations only allow asynchronous requests, in order to avoid blocking the UI thread. If on a UI thread, you will need to use async programming in F# (see Chapter 11). If already on a background thread, you can make a synchronous request like this:

```
Async.FromBeginEnd(q.BeginExecute, q.EndExecute) |> Async.RunSynchronously
```

Paginated requests are a little harder; you will need a similarly modified version of the pagination code shown later in this chapter.

Handling Pagination in OData

As mentioned in the side bar above, OData service implementations usually only return the first 20 to 100 results of a query. To get the remaining results, you need to use "pagination." This is a common pattern for all service-oriented programming, but is particularly tricky for F#. Here is a function you can use to collect paginated results into a single sequence:

```
open System.Data.Services.Client
open System.Linq

let executePaginated (ctxt: DataServiceContext) (query: IQueryable<'T>) =
    match query with
    | :? DataServiceQuery<'T> as q ->
        seq {
            let rec loop (cont: DataServiceQueryContinuation<'T>) = seq {
                if cont <> null then
                    let rsp = ctxt.Execute cont
                    yield! rsp
                    yield! loop (rsp.GetContinuation())}
            let rsp = q.Execute()
```

```
            yield! rsp
            let cont = (rsp :?> QueryOperationResponse<'T>).GetContinuation()
            yield! loop cont }
    | _ -> query.AsEnumerable()
```

You use it like this:

```
let allCustomersQuery =
    query { for c in db.Customers do select c }
    |> executePaginated db.DataContext
    |> Seq.toList
```

Example - Language Integrated SQL

For our second example of working with data through a type provider, we use data drawn from a relational database and queried using SQL. The following sections show how to perform relational database queries using F# 3.0 queries. F# 3.0 uses F# quotation metaprogramming to represent SQL queries. These are then translated across to SQL and executed using the Microsoft LINQ libraries that are part of .NET Framework 4.0 or higher.

We assume you're working with the `Northwnd.mdf` database, a common database used in many database samples. You can download this sample database as part of the F# Power Pack, or from many other sources on the Web.

The code to access the database is very simple and remarkably similar to that used to access OData:

```
#r "System.Data.Linq.dll"
#r "FSharp.Data.TypeProviders.dll"

open Microsoft.FSharp.Linq
open Microsoft.FSharp.Data.TypeProviders

type NorthwndDb =
    SqlDataConnection<ConnectionString =
                        @"AttachDBFileName  = 'C:\Scripts\northwnd.mdf';
                          Server='.\SQLEXPRESS';User Instance=true;Integrated Security=SSPI",
                      Pluralize=true>

let db = NorthwndDb.GetDataContext()
```

In this code, you first reference two libraries: the "runtime" library for SQL data access `System.Data.Linq`, and the F# library for the F# 3.0 SQL type providers called `FSharp.Data.TypeProviders.dll`. You then create an instance of a "data context" to access the database. You can now get data from the service as follows:

```
let customersSortedByCountry =
    query { for c in db.Customers do
            sortBy c.Country
            select (c.Country, c.CompanyName) }
    |> Seq.toList
```

When run, this gives the full list of customers from the database:

```
val customersSortedByCountry : (string * string) list =
  [("Argentina", "Cactus Comidas para llevar");
   ("Argentina", "Océano Atlántico Ltda."); ("Argentina", "Rancho grande");
   ...
   ("Venezuela", "LINO-Delicateses"); ("Venezuela", "HILARION-Abastos");
   ("Venezuela", "GROSELLA-Restaurante")]
```

To understand what's going on, it's helpful to add the following line before executing the code above:

```
db.DataContext.Log <- System.Console.Out
```

After you add this, you will see that the output begins:

```
SELECT [t0].[Country] AS [Item1], [t0].[ContactName] AS [Item2]
FROM [dbo].[Customers] AS [t0]
ORDER BY [t0].[Country]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.17929
```

As you can see, the query has been converted into SQL text and sent to the relational database for execution. A slightly more complex query is shown below:

```
let selectedEmployees =
    query { for emp in db.Employees do
            where (emp.BirthDate.Value.Year > 1960)
            where (emp.LastName.StartsWith "S")
            select (emp.FirstName, emp.LastName)
            take 5 }
    |> Seq.toList
```

The results are as follows (only one employee is ultimately selected):

```
SELECT TOP (5) [t0].[FirstName] AS [Item1], [t0].[LastName] AS [Item2]
FROM [dbo].[Employees] AS [t0]
WHERE ([t0].[LastName] LIKE @p0) AND (DATEPART(Year, [t0].[BirthDate]) > @p1)
-- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [S%]
-- @p1: Input Int (Size = -1; Prec = 0; Scale = 0) [1960]
-- Context: SqlProvider(Sql2008) Model: AttributedMetaModel Build: 4.0.30319.17929

val selectedEmployees : (string * string) list = [("Michael", "Suyama")]
```

More on Queries

In this section, you will learn how to perform a number of relational query operations in the F# query syntax.

Sorting

The sorting operations in queries are `sortBy`, `sortByDescending`, `thenBy`, `thenByDescending`, and their corresponding versions for nullable values such as `sortByNullable`. These are added to a query starting with one of the `sortBy` operations (for the first sorting key) and a sequence of `thenBy` operations (for secondary and subsequent sorting keys). Sorting is done in ascending order unless the "descending" variations are used. For example:

```
let customersSortedTwoColumns =
    query { for c in db.Customers do
            sortBy c.Country
            thenBy c.Region
            select (c.Country, c.Region, c.CompanyName) }
    |> Seq.toList
```

Giving:

```
val customersSortedTwoColumns : (string * string * string) list =
  [("Argentina", null, "Cactus Comidas para llevar");
   ("Argentina", null, "Océano Atlántico Ltda.");
   ("Argentina", null, "Rancho grande"); ("Austria", null, "Piccolo und mehr");
   ("Austria", null, "Ernst Handel"); ("Belgium", null, "Maison Dewey");
   ...
   ("Venezuela", "DF", "GROSELLA-Restaurante");
   ("Venezuela", "Lara", "LILA-Supermercado");
   ("Venezuela", "Nueva Esparta", "LINO-Delicateses");
   ("Venezuela", "Táchira", "HILARION-Abastos")]
```

Aggregation

Aggregation operations (sum, average, maximum, minimum) are performed by using the query operators `sumBy`, `sumByNullable`, `averageBy`, `averageByNullable`, `maxBy`, `maxByNullable`, `minBy` or `minByNullable`. For example:

```
let totalOrderQuantity =
    query { for order in db.OrderDetails do
            sumBy (int order.Quantity) }

let customersAverageOrders =
    query { for c in db.Customers do
            averageBy (float c.Orders.Count) }
```

Giving:

```
val totalOrderQuantity : int = 51317
val customersAverageOrders : float = 9.120879121
```

Nullables

Data in SQL tables, OData services and other data sources may be missing values. When using these data sources via the usual type providers or code generators for F# (e.g. `ODataService`, `SqlDataProvider` and `SqlEntityProvider`), the potential for missing primitive values is represented using the .NET type `System.Nullable<_>`.

Full techniques to handle nullable values are discussed in the MSDN documentation for F# queries. Here we show just some examples. For example, to compute the average price over the range of products on offer, you can use `averageByNullable`:

```
let averagePriceOverProductRange =
    query { for p in db.Products do
            averageByNullable p.UnitPrice }
```

Giving:

```
val averagePriceOverProductRange : System.Nullable<decimal> = 28.8663M
```

Note that the result of the averaging is itself a nullable value.

Working with nullable values can be slightly awkward in F#. Here are some techniques to use:

- Very often, you will need to eliminate nullable values using either the `.Value` property, or the `.GetValueOrDefault()` method. You can also call conversion functions such `int` or `float`.
- To create a nullable value, use the constructor for the System.Nullable type, e.g., `System.Nullable(3)`.
- To compare, add, subtract, multiply, or divide nullable values, use the special operators such as `?+?` in `Microsoft.FSharp.Linq.NullableOperators`.

Inner Queries

It is very common to use one or more "inner" queries to collect information about an entity before selecting it. For example, consider the task of writing a query which iterates the customers, sums the number of orders (if any), and finds the average unit price of orders (if any) for that customer. Here is the query:

```
let totalOrderQuantity =
    query { for c in db.Customers do
            let numOrders =
                query { for o in c.Orders do
                        for od in o.OrderDetails do
                        sumByNullable (Nullable(int od.Quantity)) }
            let averagePrice =
                query { for o in c.Orders do
                        for od in o.OrderDetails do
                        averageByNullable (Nullable(od.UnitPrice)) }
            select (c.ContactName, numOrders, averagePrice) }
    |> Seq.toList
```

Giving:

```
val totalOrderQuantity :
  (string * System.Nullable<int> * System.Nullable<decimal>) list =
  [("Maria Anders", 174, 26.7375M); ("Ana Trujillo", 63, 21.5050M);
   ("Antonio Moreno", 359, 21.7194M); ("Thomas Hardy", 650, 19.1766M);
   ...
   ("Zbyszek Piestrzeniewicz", 205, 20.6312M)]
```

Grouping

The previous example showed how to use inner queries to compute and return a number of statistical properties of an entity using a query. Very often, this is instead done over a group of entities. To group entities, you use the `groupBy .. into …` operator:

```
let productsGroupedByNameAndCountedTest1 =
    query { for p in db.Products do
            groupBy p.Category.CategoryName into group
            let sum =
                query { for p in group do
                        sumBy (int p.UnitsInStock.Value) }
            select (group.Key, sum) }
    |> Seq.toList
```

Giving:

```
val productsGroupedByNameAndCountedTest1 : (string * int) list =
  [("Beverages", 559); ("Condiments", 507); ("Confections", 386);
   ("Dairy Products", 393); ("Grains/Cereals", 308); ("Meat/Poultry", 165);
   ("Produce", 100); ("Seafood", 701)]
```

Joins

You can write both normal SQL joins and "group" joins using F# queries. For example, a simple join is written as follows:

```
let innerJoinQuery =
    query { for c in db.Categories do
            join p in db.Products on (c.CategoryID =? p.CategoryID)
            select (p.ProductName, c.CategoryName) } //produces flat sequence
    |> Seq.toList
```

The operator `=?` is a nullable comparison operator, mentioned in the previous section, because `p.CategoryID` may be missing. This gives:

```
val innerJoinQuery : (string * string) list =
  [("Chai", "Beverages"); ("Chang", "Beverages");
   ("Aniseed Syrup", "Condiments");
   ...
   ("Lakkalikööri", "Beverages");
   ("Original Frankfurter grüne Soße", "Condiments")]
```

As it happens, many joins are actually implicit in F# queries over types provided by the usual type providers. For example, the above query could have been written:

```
let innerJoinQuery =
    query { for p in db.Products do
            select (p.ProductName, p.CategoryName) }
    |> Seq.toList
```

Likewise, you can also write "group joins", where the inner iteration results in an overall group of elements that satisfy the joining constraint. For example:

```
    let innerGroupJoinQueryWithAggregation =
        query { for c in db.Categories do
                groupJoin p in db.Products on (c.CategoryID =? p.CategoryID) into prodGroup
                let groupMax = query { for p in prodGroup do maxByNullable p.UnitsOnOrder }
                select (c.CategoryName, groupMax) }
    |> Seq.toList
```

This gives:

```
val innerGroupJoinQueryWithAggregation : (string * Nullable<int16>) list =
  [("Beverages", 40s); ("Condiments", 100s); ("Confections", 70s);
   ("Dairy Products", 70s); ("Grains/Cereals", 80s); ("Meat/Poultry", 0s);
   ("Produce", 20s); ("Seafood", 70s)]
```

More on Relational Databases and ADO.NET

So far in this chapter, you have seen how to perform simple "raw" HTTP web requests to an information service, and how to use the type provider and query mechanisms to access OData and SQL data in a more directly strongly typed, and clear way.

However, it is often necessary to access databases in a "raw" way as well. To do this, you use the ADO.NET library which has been part of .NET since its first releases. Like web requests, this is a "lowest common denominator" way of doing information access from F# code. While it is usually nicer to use the type provider mechanism where possible, this is not always possible for all databases, and is also not really possible for cases where you are creating tables dynamically. Further, some data access standards (e.g. ODBC, a common data connectivity standard developed in the late 1990s) will definitely require you to use lower level libraries that look a lot like ADO.NET. Note that you can also use the newer Entity Framework to work with databases on a higher conceptual level. However, in this current discussion, we are mostly concerned with low-level details such as programmatically creating tables, executing update and insert statements, and querying using plain SQL code.

First, however, we examine databases more generally and give a guide to your choices about which database to use in the first place. Databases provide many benefits. Some of the more important ones are listed here:

- *Data security:* When you have centralized control of your data, you can erect a full security system around the data, implementing specific access rules for each type of access or parts of the database.
- *Sharing data*: Any number of applications with the appropriate access rights can connect to your database and read the data stored within—without needing to worry about containing the logic to extract this data. As you will see shortly,

applications use various query languages (most notably SQL) to communicate with databases.

- *A logical organization of data*: You can write new applications that work with the same data without having to worry about how the data is physically represented and stored. On the basic level, this logical structure is provided by a set of entities (data tables) and their relationships.
- *Avoiding data redundancy*: Having all requirements from each consuming application up front helps to identify a logical organization for your data that minimizes possible redundancy. For instance, you can use foreign keys instead of duplicating pieces of data. *Data normalization* is the process of systematically eliminating data redundancy, a large but essential topic that we don't consider in this book.
- *Transactions*: Reading from and writing to databases occurs atomically, and as a result, two concurrent transactions can never leave data in an inconsistent, inaccurate state. *Isolation levels* refer to specific measures taken to ensure transaction isolation by locking various parts of the database (fields, records, tables). Higher isolation levels increase locking overhead and can lead to a loss of parallelism by rendering concurrent transactions sequential; on the other hand, no isolation can lead to inconsistent data.
- *Maintaining data integrity*: Databases make sure data is stored accurately. Having no redundancy is one way to maintain data integrity (if a piece of data is changed, it's changed in the only place it occurs; thus, it remains accurate); on the other hand, data security and transaction isolation are needed to ensure that the data stored is modified in a controlled manner.

Table 13-1 shows some of the most common database engines, all of which can be used from F# and .NET.

Table 13-1. *Common Databases*

Name	Type	Description	Available From
PostgreSQL	Open source	Open source database engine	`www.postgresql.org`
SQLite	Open source	Small, embeddable, zero-configuration SQL database engine	`www.sqlite.org`
DB2	Commercial	IBM's database engine	`www-01.ibm.com/software/data/db2/ad/dotnet.html`
Firebird	Open source	Based on Borland Interbase	`www.firebirdsql.org`
MySQL	Open source	Reliable and popular database	`www.mysql.com`
Mimer SQL	Commercial	Reliable database engine	`www.mimer.com`
Oracle	Commercial	One of the most popular enterprise database engines	`www.oracle.com`
SQL Server	Commercial	Microsoft's main database engine	`www.microsoft.com/sql`
SQL Server Express	Commercial	Free and easy-to-use version of SQL Server	`www.microsoft.com/express/database`
Sybase iAnywhere	Commercial	Mobile database engine	`www.ianywhere.com`

Applications communicate with relational databases using Structured Query Language (SQL). Each time you create tables, create relationships, insert new records, or update or delete existing ones, you are explicitly or implicitly issuing SQL statements to the database. The examples in this chapter use a dialect of Standard SQL called Transact-SQL (T-SQL), used by SQL Server and SQL Server Express. SQL has syntax to define the structure of a database schema (loosely speaking, a collection of data tables and their relations) and also syntax to manage the data within. These subsets of SQL are called *Data Definition Language* (DDL) and *Data Manipulation Language* (DML), respectively.

ADO.NET is the underlying database-access machinery in the .NET Framework, and it provides full XML support, disconnected and typed datasets, scalability, and high performance. This section gives a brief overview of the ADO.NET fundamentals.

With ADO.NET, data is acquired through a *connection* to the database via a provider. This connection serves as a medium against which to execute a *command*; this can be used to fetch, update, insert, or delete data from the data store. Statements and queries are articulated as SQL text (`CREATE`, `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements) and are passed to the command object's constructor. When you execute these statements, you obtain data (in the case of queries) or the number of affected rows (in the case of `UPDATE`, `INSERT`, and `DELETE` statements). The data returned can be processed via two main mechanisms: sequentially in a read-only fashion using a `DataReader` object or by loading it into an in-memory representation (a `DataSet` object) for further disconnected processing. `DataSet` objects store data in a set of table objects along with metadata that describes their relationships and constraints in a fully contained model.

Establishing Connections using ADO.NET

Before you can do any work with a database, you need to establish a connection to it. For instance, you can connect to a locally running instance of SQL Server Express using the following code:

```
open System.Data
open System.Data.SqlClient

let connString = @"Server='.\SQLEXPRESS';Integrated Security=SSPI"
let conn = new SqlConnection(connString)
```

The value `connString` is a connection string. Regardless of how you created your connection object, to execute any updates or queries on it, you need to open it first:

```
> conn.Open();;
```

If this command fails, then you may need to do one of the following:

- Consult the latest SQL Server Express samples for alternative connection strings.
- Add `UserInstance='true'` to the connection string. This starts the database engine as a user-level process.
- Change the connection string if you have a different database engine installed and running (for instance, if you're using SQL Server instead of SQL Server Express).

Connections established using the same connection string are pooled and reused depending on your database engine. Connections are often a limited resource and should generally be closed as soon as possible within your application.

Creating a Database using ADO.NET

Now that you've established a connection to the database engine, you can explicitly create a database from F# code by executing a SQL statement directly. For example, you can create a database called `company` as follows:

```
open System.Data
open System.Data.SqlClient

let execNonQuery conn s =
    let comm = new SqlCommand(s, conn, CommandTimeout = 10)
    comm.ExecuteNonQuery() |> ignore

execNonQuery conn "CREATE DATABASE company"
```

You use `execNonQuery` in the subsequent sections. This method takes a connection object and a SQL string and executes it as a SQL command, ignoring its result.

■ **Note** If you try to create the same database twice, you receive a runtime exception. However, if you intend to drop an existing database, you can do so by issuing a `DROP DATABASE company` SQL command. The `DROP` command can also be used for other database artifacts, including tables, views, and stored procedures.

Creating Tables using ADO.NET

You can execute a simple SQL command to create a table; all you need to do is specify its data fields and their types and whether null values are allowed. The following example creates an `Employees` table with a primary key `EmpID` and `FirstName`, `LastName`, and `Birthday` fields:

```
execNonQuery conn "CREATE TABLE Employees (
   EmpID int NOT NULL,
   FirstName varchar(50) NOT NULL,
   LastName varchar(50) NOT NULL,
   Birthday datetime,
   PRIMARY KEY (EmpID))"
```

You can now insert two new records as follows:

```
execNonQuery conn "INSERT INTO Employees (EmpId, FirstName, LastName, Birthday)
   VALUES (1001, 'Joe', 'Smith', '02/14/1965')"

execNonQuery conn "INSERT INTO Employees (EmpId, FirstName, LastName, Birthday)
   VALUES (1002, 'Mary', 'Jones', '09/15/1985')"
```

and retrieve two columns of what was inserted using a fresh connection and a data reader:

```
let query() =
    seq {
        use conn = new SqlConnection(connString)
        conn.Open()
        use comm = new SqlCommand("SELECT FirstName, Birthday FROM Employees", conn)
        use reader = comm.ExecuteReader()
        while reader.Read() do
```

```
            yield (reader.GetString 0, reader.GetDateTime 1)
    }
```

When you evaluate the query expression in F# Interactive, a connection to the database is created and opened, the command is built, and the reader is used to read successive elements:

```
> fsi.AddPrinter(fun (d: System.DateTime) -> d.ToString());;

> query();;

val it : seq<string * System.DateTime> =
    seq [("Joe", 14/02/1965 12:00:00AM); ("Mary", 15/09/1985 12:00:00AM)]
```

The definition of `query` uses sequence expressions that locally define new `IDisposable` objects such as `conn`, `comm`, and `reader` using declarations of the form `use var = expr`. These ensure that the locally defined connection, command, and reader objects are disposed after exhausting the entire sequence. See Chapters 4, 8, and 9 for more details about sequence expressions of this kind.

F# sequences are on-demand (that is, lazy), and the definition of `query` doesn't open a connection to the database. This is done when the sequence is first iterated; a connection is maintained until the sequence is exhausted.

Note that the command object's `ExecuteReader` method returns a `DataReader` instance that is used to extract the typed data returned from the query. You can read from the resulting sequence in a straightforward manner using a sequence iterator. For instance, you can use a simple anonymous function to print data on the screen:

```
> query() |> Seq.iter (fun (fn, bday) -> printfn "%s has birthday %O" fn bday);;

Joe has birthday 14/02/1965 00:00:00
Mary has birthday 15/09/1985 00:00:00
```

The query brings the data from the database in-memory, although still as a lazy sequence. You can then use standard F# in-memory data transformations on the result:

```
> query()
  |> Seq.filter (fun (nm, bday) -> bday < System.DateTime.Parse("01/01/1985"))
  |> Seq.length;;

val it : int = 1
```

However, be aware that these additional transformations are happening in-memory and not in the database.

The command object has different methods for executing different queries. For instance, if you have a statement, you need to use the `ExecuteNonQuery` method (for `UPDATE`, `INSERT`, and `DELETE` statements, as previously in `execNonQuery`), which returns the number of rows affected (updated, inserted, or deleted), or the `ExecuteScalar` method, which returns the first column of the first row of the result, providing a fast and efficient way to extract a single value, such as the number of rows in a table or a result set.

In the previous command, you extracted fields from the result rows using `Get`*`XXX`* methods on the `reader` object. The particular methods have to match the types of the fields selected in the SQL query, and

a mismatch results in a runtime `InvalidCastException`. For these and other reasons, `DataReader` tends to be suitable only in situations when the following items are true:

- You need to read data only in a sequential order (as returned from the database). `DataReader` provides forward-only data access.
- The field types of the result are known, and the query isn't configurable.
- You're reading only and not writing data. `DataReader` provides read-only access.
- Your use of the `DataReader` is localized. The data connection is open throughout the reader loop.

Database connections are precious resources, and you should always release them as soon as possible. In the previous case, you did this by using a locally defined connection. It's also sufficient to implicitly close the reader by constructing it with the `CloseConnection` option that causes it to release and close the data connection upon closing the reader instance.

Common options include `SchemaOnly`, which you can use to extract field information only (without any data returned); `SingleResult` to extract a single value only (the same as using the `ExecuteScalar` method discussed earlier); `SingleRow` to extract a single row; and `KeyInfo` to extract additional columns (appended to the end of the selected ones) automatically that uniquely identify the rows returned.

Using Stored Procedures via ADO.NET

Stored procedures are defined and stored in your relational database and provide a number of benefits over literal SQL. First, they're external to the application and thus provide a clear division of the data logic from the rest of the application. This enables you to make data-related modifications without having to change application code or having to redeploy the application. Second, they're stored in the database in a prepared or compiled form and thus are executed more efficiently than literal SQL statements (although those can be prepared as well at a one-time cost, they're still contained in application space, which is undesirable). Supplying arguments to stored procedures instantiates the compiled formula.

In Visual Studio, you can add stored procedures just like any other database artifacts using the Server Explorer window: right-click the Stored Procedures item in the appropriate database, and select Add New Stored Procedure. Doing so creates a stored procedure template that you can easily customize. Alternatively, you can add stored procedures programmatically using the `CREATE PROCEDURE` SQL command. Consider the following stored procedure that returns the first and last names of all employees whose last name matches the given pattern:

```
execNonQuery conn "
CREATE PROCEDURE dbo.GetEmployeesByLastName ( @Name nvarchar(50) ) AS
    SELECT Employees.FirstName, Employees.LastName
    FROM Employees
    WHERE Employees.LastName LIKE @Name"
```

You can wrap this stored procedure in a function as follows:

```
let GetEmployeesByLastName (name: string) =
    use comm = new SqlCommand("GetEmployeesByLastName", conn,
                              CommandType = CommandType.StoredProcedure)
    comm.Parameters.AddWithValue("@Name", name) |> ignore
    use adapter = new SqlDataAdapter(comm)
    let table = new DataTable()
    adapter.Fill(table) |> ignore
    table
```

You can execute the stored procedure as follows to find employees with the last name Smith:

```
> for row in GetEmployeesByLastName("Smith").Rows do
     printfn "row = %O, %O" (row.Item "FirstName") (row.Item "LastName");;

row = Joe, Smith
```

Using WSDL Services

Since around 2000, a popular web service protocol has been WSDL. The use of this protocol is now declining, but it is still common to find useful WSDL web service implementations, particularly in enterprises. In this section, you learn how to access a WSDL service from F#.

WSDL web services can be easily consumed in F# in much the same way as OData web services by using the `WsdlService` type provider that comes with F#. For example, consider the weather service at `http://www.webservicex.com/globalweather.asmx?wsdl`. This can be accessed using the following code:

```
#r "System.ServiceModel.dll"
#r "FSharp.Data.TypeProviders.dll"

open Microsoft.FSharp.Data.TypeProviders

type Weather = WsdlService<"http://www.webservicex.com/globalweather.asmx?wsdl">

let ws = Weather.GetGlobalWeatherSoap();;

let weatherInCanberra = ws.GetWeather("Canberra", "Australia")
```

In this case, the result is just XML giving the weather at the given location:

```
val weatherInCanberra : string =
  "<?xml version="1.0" encoding="utf-16"?>
<CurrentWeather>
  <Location>Canberra, Australia (YSCB) 35-18S 149-11E 580M</Location>
  <Time>Aug 14, 2012 - 12:00 PM EDT / 2012.08.14 1600 UTC</Time>
  <Wind> from the E (080 degrees) at 7 MPH (6 KT):0</Wind>
  <Visibility> greater than 7 mile(s):0</Visibility>
  <SkyConditions> partly cloudy</SkyConditions>
  <Temperature> 35 F (2 C)</Temperature>
  <Wind>Windchill: 28 F (-2 C):1</Wind>
  <DewPoint> 33 F (1 C)</DewPoint>
  <RelativeHumidity> 93%</RelativeHumidity>
  <Pressure> 30.00 in. Hg (1016 hPa)</Pressure>
  <Status>Success</Status>
</CurrentWeather>"
```

Often, WSDL web services return structured, strongly typed results, and no further XML parsing is required.

Summary

In this chapter, you learned about how the growing availability of data is changing programming, forcing programmers to incorporate more data and network access code into their applications. You learned both low-level and high-level techniques for accessing a range of web, database, and service technologies. Along the way, you learned the basics of two important F# features that are used for data access: F# queries and F# type providers. Together these give an elegant and direct way of integrating data into your programs. You also learned low-level techniques for REST requests and database access with ADO.NET. These are pragmatic techniques that allow you to do more, but less directly.

The next chapter continues on the theme of web programming by looking at a range of topics in delivering content via the Web, from delivering HTML directly to writing full web applications using WebSharper, the web application framework for F#.