David Chappell

May 2011

INTRODUCING ODATA

DATA ACCESS FOR THE WEB, THE CLOUD, MOBILE DEVICES, AND MORE

Sponsored by Microsoft Corporation

Contents

Describing OData 3

The Problem: Accessing Diverse Data in a Common Way 3

The Solution: What OData Provides 3

How OData Works: Technology Basics 5

Using OData: Example Scenarios 6

Accessing Application Data from Mobile Phones and Web Browsers 6

Exposing Data from a Cloud Application 8

Using Diverse Data Sources with Different BI Tools 9

Examining OData: A Closer Look at the Technology and Its Implementation 10

The OData Data Model 10

The OData Protocol 12

Protocol Basics 13

Serializing Data with Atom/AtomPub 13

Serializing Data with JSON 19

Issuing Queries 20

A Perspective: OData in a SOA World 21

OData Client Libraries 22

OData Services 22

Conclusion 23

For Further Reading 24

About the Author 24

Describing OData

Our world is awash in data. Vast amounts exist today, and more is created every year. Yet data has value only if it can be used, and it can be used only if it can be accessed by applications and the people who use them.

Allowing this kind of broad access to data is the goal of the *Open Data Protocol,* commonly called just *OData*. This paper provides an introduction to OData, describing what it is and how it can be applied. The goal is to illustrate why OData is important and how your organization might use it.

The Problem: Accessing Diverse Data in a Common Way

There are many possible sources of data. Applications collect and maintain information in databases, organizations store data in the cloud, and many firms make a business out of selling data. And just as there are many data sources, there are many possible clients: Web browsers, apps on mobile devices, business intelligence (BI) tools, and more. How can this varied set of clients access these diverse data sources?

One solution is for every data source to define its own approach to exposing data. While this would work, it leads to some ugly problems. First, it requires every client to contain unique code for each data source it will access, a burden for the people who write those clients. Just as important, it requires the creators of each data source to specify and implement their own approach to getting at their data, making each one reinvent this wheel. And with custom solutions on both sides, there's no way to create an effective set of tools to make life easier for the people who build clients and data sources.

Thinking about some typical problems illustrates why this approach isn't the best solution. Suppose a Web application wishes to expose its data to apps on mobile phones, for instance. Without some common way to do this, the Web application must implement its own idiosyncratic approach, forcing every client app developer that needs its data to support this. Or think about the need to connect various BI tools with different data sources to answer business questions. If every data source exposes data in a different way, analyzing that data with various tools is hard—an analyst can only hope that her favorite tool supports the data access mechanism she needs to get at a particular data source.

Defining a common approach makes much more sense. All that's needed is agreement on a way to model data and a protocol for accessing that data—the implementations can differ. And given the Web-oriented world we live in, it would make sense to build this technology with existing Web standards as much as possible. This is exactly the approach taken by OData.

The Solution: What OData Provides

OData defines an abstract data model and a protocol that let any client access information exposed by any data source. Figure 1 shows some of the most important examples of clients and data sources, illustrating where OData fits in the picture.

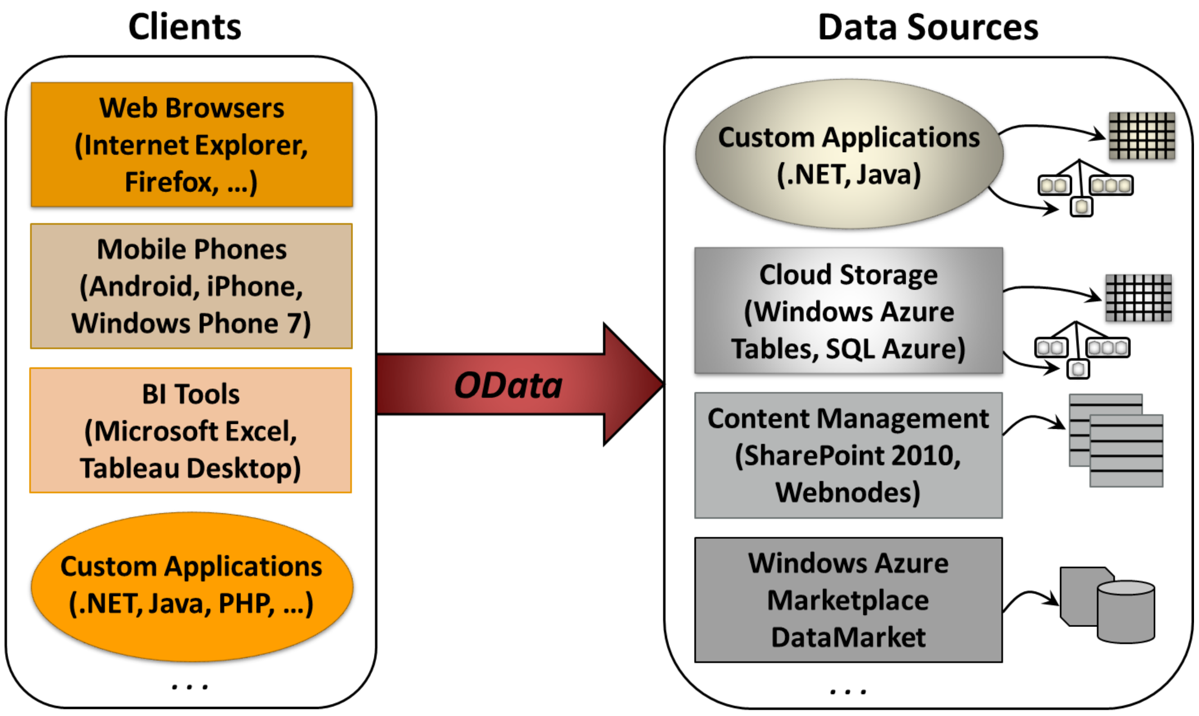
Clients
Web Browsers (Internet Explorer, Firefox, …)
Mobile Phones (Android, iPhone, Windows Phone 7)
BI Tools (Microsoft Excel, Tableau Desktop)
Custom Applications (.NET, Java, PHP, …)
. . .
OData
Data Sources
Custom Applications (.NET, Java)
Cloud Storage (Windows Azure Tables, SQL Azure)
Content Management (SharePoint 2010, Webnodes)
Windows Azure Marketplace DataMarket
. . .

Figure 1: Any OData client can access data provided by any OData data source.

As the figure illustrates, OData allows mixing and matching clients and data sources. Some of the most important examples of data sources that support OData today are:

- Custom applications: Rather than creating its own mechanism to expose data, an application can instead use OData. Facebook, Netflix, and eBay all expose some of their information via OData today, as do a number of custom enterprise applications. To make this easier to do, OData libraries are available that let .NET Framework and Java applications act as data sources.

- Cloud storage: OData is the built-in data access protocol for tables in Microsoft's Windows Azure, and it's supported for access to relational data in SQL Azure as well. Using available OData libraries, it's also possible to expose data from other cloud platforms, such as Amazon Web Services.

- Content management software: For example, SharePoint 2010 and Webnodes both have built-in support for exposing information through OData.

- Windows Azure Marketplace DataMarket: This cloud-based service for discovering, purchasing, and accessing commercially available datasets lets applications access those datasets through OData.

While it's possible to access an OData data source from an ordinary browser—the protocol is based on HTTP—client applications usually rely on a client library. As Figure 1 shows, the options supported today include:

- Web browsers: JavaScript code running inside any popular Web browser, such as Internet Explorer or Firefox, can access an OData data source. An OData client library is available for Silverlight applications as well, and other rich Internet applications can also act as OData clients.

- Mobile phones. OData client libraries are available today for Android, iOS (the operating system used by iPhones and iPads), and Windows Phone 7.
- Business intelligence tools: Microsoft Excel provides a data analysis tool called PowerPivot that has built-in support for OData. Other desktop BI tools also support OData today, such as Tableau Software's Tableau Desktop.
- Custom applications: Business logic running on servers can act as an OData client. Support is available today for code created using the .NET Framework, Java, PHP, and other technologies.

The fundamental idea is that any OData client can access any OData data source. Rather than creating unique ways to expose and access data, data sources and their clients can instead rely on the single solution that OData provides.

OData was originally created by Microsoft. Yet while several of the examples in Figure 1 use Microsoft technologies, OData isn't a Microsoft-only technology. In fact, Microsoft has included OData under its Open Specification Promise, guaranteeing the protocol's long-term availability for others. While much of today's OData support is provided by Microsoft, it's more accurate to view OData as a general purpose data access technology that can be used with many languages and many platforms.

How OData Works: Technology Basics

Providing a way for all kinds of clients to access all kinds of data is clearly a good thing. But what's needed to make the idea work? Figure 2 shows the fundamental components of the OData technology family.

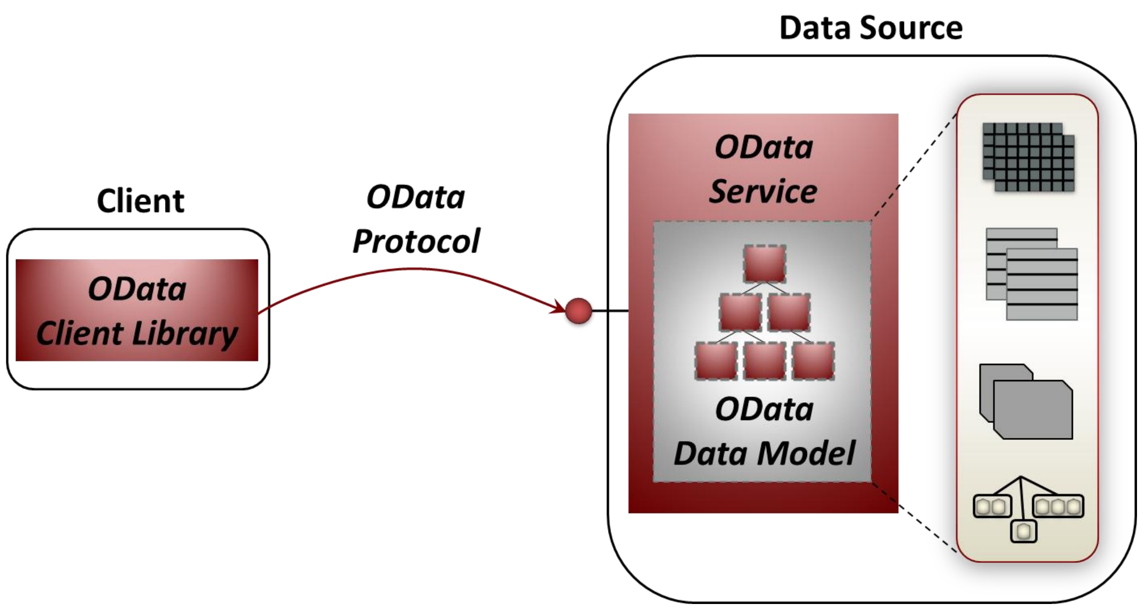

Figure 2: An OData service exposes data via the OData data model, which clients access with an OData client library and the OData protocol.

The OData technology has four main parts:

- The *OData data model*, which provides a generic way to organize and describe data. OData uses the Entity Data Model (EDM), the same approach that's used by Microsoft's Entity Framework (EF)[1].

- The *OData protocol*, which lets a client make requests to and get responses from an OData service. At bottom, the OData protocol is a set of RESTful interactions—it's just HTTP. Those interactions include the usual create/read/update/delete (CRUD) operations, along with an OData-defined query language. Data sent by an OData service can be represented on the wire today either in the XML-based format defined by Atom/AtomPub or in JavaScript Object Notation (JSON).

- *OData client libraries*, which make it easier to create software that accesses data via the OData protocol. Because OData relies on REST, using an OData-specific client library isn't strictly required. But most OData clients are applications, and so providing pre-built libraries for making OData requests and getting results makes life simpler for the developers who create those applications.

- An *OData service*, which exposes an endpoint that allows access to data. This service implements the OData protocol, and it also uses the abstractions of the OData data model to translate data between its underlying form, which might be relational tables, SharePoint lists, or something else, into the format sent to the client.

Given this basic grasp of the OData technology, it's possible to get a better sense of how it can be used. The best way to do this is to look at some representative OData scenarios.

Using OData: Example Scenarios

Because OData is a general-purpose data access mechanism, it can be used in many different ways. This section looks at three representative examples:

- Using OData to let mobile phones and Web browsers access a custom application's data.

- Letting application business logic use OData to access data exposed in the cloud.

- Allowing different BI tools to access diverse data sources through OData.

Accessing Application Data from Mobile Phones and Web Browsers

Users commonly access Web applications today through browsers. More and more, however, custom client applications are used in place of browsers, especially on mobile devices. And when those client apps need to access a Web application's functionality, using standard REST calls can work well.

But exposing data is harder. Without conventions for doing this, the creator of a Web application needs to create a data model (since he probably doesn't want to expose his application's internal database structure to the world), a query language (to allow more than just simple reads), client libraries (to help diverse clients access the data), and perhaps even tools (to help people create those clients).

[1] Even though the EDM was originally created as part of Entity Framework, OData borrows just the EDM modeling aspect of EF. An implementation of OData isn't required to use EF itself.

If the application instead exposes its data through OData, life gets significantly simpler—these things are already available. Figure 3 illustrates this, showing how a custom application can use OData to expose its data both to client apps on mobile phones and to Web browsers.

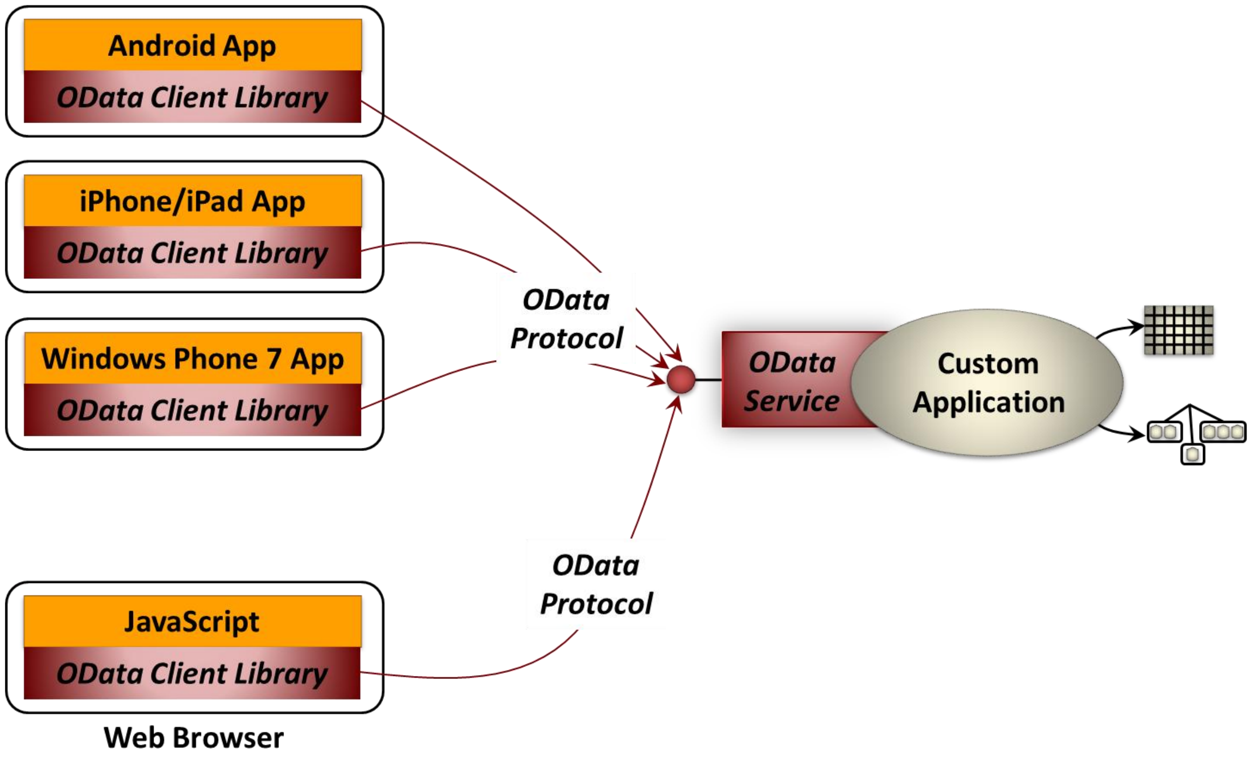

Figure 3: Mobile phones and Web browsers can use OData to access data exposed by a custom application.

In this example, data exposed by a Web application is accessed by client apps running on three different kinds of mobile devices: Android, iPhone/iPad, and Windows Phone 7. All three rely on OData client libraries made available by Microsoft today, and all three see the same data model exposed by the custom application's OData service. When requesting data via the OData protocol, each application can choose the format it wants that data delivered in: XML with Atom/AtomPub or JSON.

Similarly, JavaScript code running in a Web browser uses another Microsoft-provided OData client library to access the application's data. The data is exposed using the same data model, and it's accessed via the same OData protocol. Because the client is written in JavaScript, it probably elects to have data delivered to it in JSON (although this isn't required). And although it's not shown in the figure, Microsoft also provides an OData client library for Silverlight, supporting more functional browser applications.

It's important to understand that nothing about OData requires an application to expose the structure of its internal data to clients. A client only sees the data model provided by the OData service, not the raw underlying data. How the application maps its data to the OData data model is entirely up to the developer. If the underlying data source is relational tables, for example, she might choose to reflect one or more tables directly in her application's OData data model, but perhaps omit some of columns in these tables. Alternatively, she might create an entirely different mapping where the OData data model is quite different from the underlying database.

Whatever choice she makes, the OData service is free to interpose logic, such as rules for access control, between clients and that data. Using OData needn't mean that clients can see directly into the structure of an application's data.

It's also important to understand that OData is designed to protect data sources from clients that request too much data. As long as clients request small amounts of data, this problem doesn't arise. But suppose a client requests all of the data in, say, a relational table—what then? Is the OData service obligated to return everything in a single response? The answer is no. Instead, a service is free to define its own page size, then return data a page at a time. It can also provide a continuation indicator, letting the client request the next page. Because of this, a client request for a large amount of data needn't overwhelm an OData service's ability to deal with it.

Exposing Data from a Cloud Application

Cloud platforms are changing how we build and run applications. They're also changing how we store and access data. OData can play a role in these changes.

For example, think about a firm that sells products directly to customers via the Web. Suppose this firm also wishes to let partner organizations access its product information from their own applications. To do this, the firm might build an application and store its data on a cloud platform, such as the Windows Azure platform. This cloud application will interact with users via browsers as usual. It can also use OData to expose the application's data to software created by its partners. Figure 4 shows how this looks.

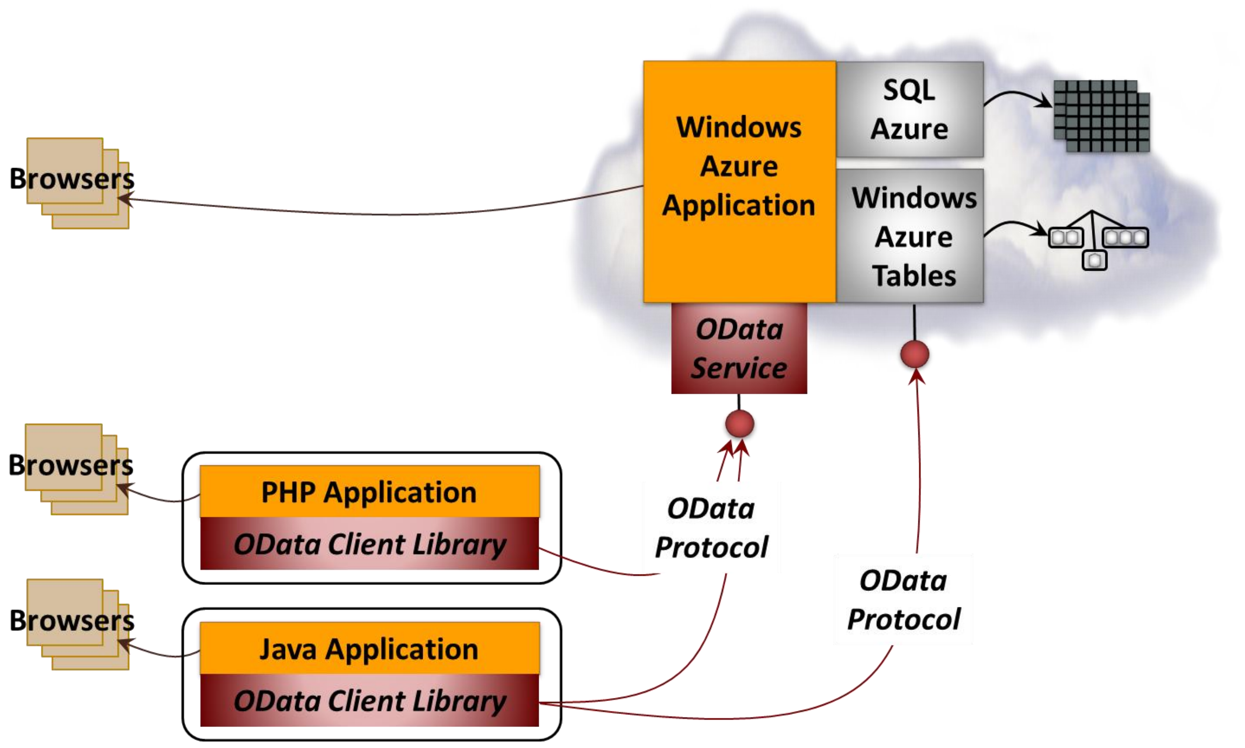

Figure 4: Diverse applications can use OData to access data stored in the cloud.

As the figure shows, the Windows Azure application interacts with customers via browsers. This application might be built using the .NET Framework or Java or something else—Windows Azure supports several options. Whatever language it's in, the application can expose an OData service to provide external access to its data. In this example, partners have created applications using PHP and Java, both of which have OData client libraries available. These partner applications then interact with their own users through browsers or perhaps in some other way, accessing the cloud data as needed. This approach, with an application providing a standard browser interface while also exposing its data to other applications, is a common way to use OData today.

A partner application can also use OData to access information that the cloud application stores in Windows Azure tables, as Figure 4 illustrates. OData is the native access protocol for Windows Azure tables, and as long as it's authorized to do so, another application can work directly with this information. It's more common today to expose an OData service from an application rather than directly from a data store, but both approaches are possible.

Using Diverse Data Sources with Different BI Tools

Business intelligence, analyzing information to extract meaning, is an important part of how people use data. Analyzing data first requires accessing data, and given the multiplicity of BI tools and data sources in use today, this is a non-trivial problem. Different analysts prefer different tools, and data is kept in different forms in different places. Much of an organization's useful data is likely to be wrapped inside custom and packaged applications, for example, while many organizations also keep useful business data in SharePoint lists. Another possible source for data is Microsoft's Windows Azure Marketplace DataMarket, which provides a cloud-based way to purchase and access commercial data sets.

Suppose an analyst wishes to combine data from these various sources. Maybe a retailer is trying to decide where to locate a new store, for example, and so needs to look at sales information from one of its custom applications, customer survey data stored in SharePoint lists, and demographic data acquired from DataMarket. Or perhaps analysts in a local government wish to access emergency call data from the city's custom call center application, police reports stored in SharePoint, and national crime statistics available through DataMarket. In both cases, it's entirely possible that different analysts wish to use different tools to work with this data.

The problem is clear: How can we connect multiple clients to multiple data sources? Without a common approach to exposing and accessing data, the situation is bleak. OData can help, as Figure 5 shows.

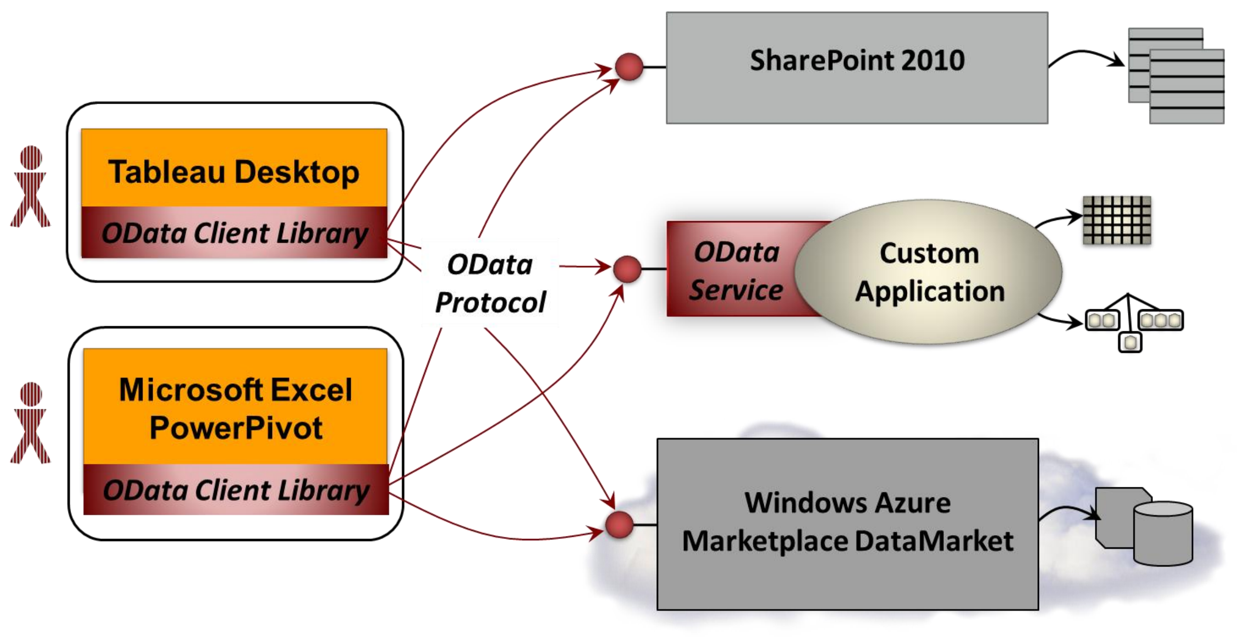

Figure 5: Different BI tools can use OData to access data stored in different formats across different data sources.

In this example, two different analysts using different BI tools—Tableau Desktop and Microsoft Excel's PowerPivot—are accessing data from the three data sources just listed: SharePoint 2010 lists, a custom application, and Windows Azure Marketplace DataMarket. All of these technologies can use OData today, and so making these connections is straightforward. Because clients and data sources speak the common language of OData, hooking them together gets simpler, and analysts can begin working with new data more rapidly.

Examining OData: A Closer Look at the Technology and Its Implementation

OData began life as a Microsoft project code-named *Astoria*. The technology was then renamed *ADO.NET Data Services* before its protocol and data model were separated out and became OData. (The parts of ADO.NET Data Services that were focused on the Windows implementation of OData are now known as *WCF Data Services*.) Whatever the name, though, the fundamental technology of OData has remained the same.

As described earlier, it's useful to think about the OData world in four parts: the data model, the protocol, the client libraries, and the OData service itself. This section describes all four, beginning with the data model.

The OData Data Model

To provide a general way for any client to access any kind of information, OData provides an abstract data model. Yet data comes in many different forms, and it can be related to other data in a variety of ways. How can a single data model encompass this diversity?

OData's answer is the Entity Data Model. In many ways a modern take on the familiar entity-relationship model, the EDM models data as *entities* and *associations* among those entities. This general approach lets the EDM—and thus OData—work with pretty much any kind of data. Figure 6 illustrates the fundamentals of how the EDM describes data.

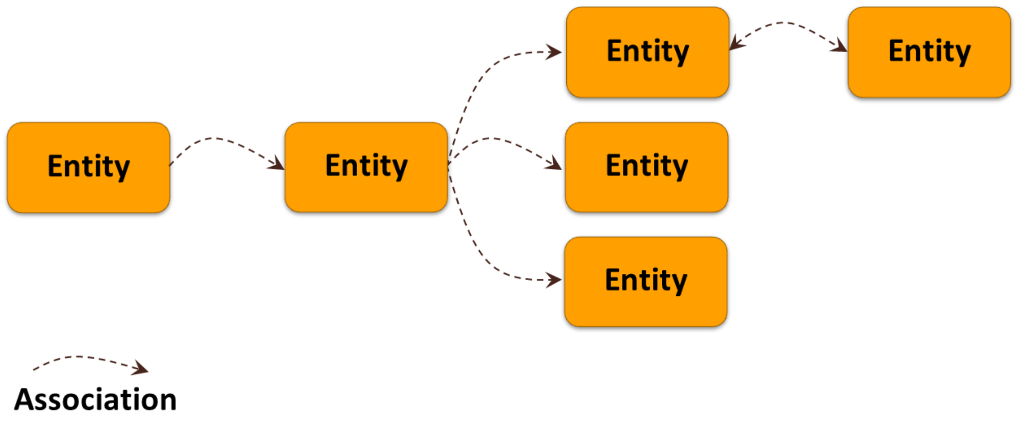

Figure 6: The Entity Data Model describes data as entities connected by associations.

As the figure shows, associations between entities can be one-to-one or many-to-one. An association can also be unidirectional, as are most of those shown here, or bi-directional, like the association in the upper right. Whatever structure is used, it's important to understand that the EDM describes only the logical structure of data. How that data is stored physically is irrelevant.

The data exposed by an OData service can come from many sources, and how this data is mapped to the EDM is up to the creator of that service. For example, an OData service exposing relational data might represent each table as an entity, with foreign key relationships among those tables modeled as associations. A service that's exposing data directly from a set of Java objects might model each object as an entity and the connections among objects as associations.

There's more to the EDM than just entities and associations, however. Figure 7 shows a more complete picture.

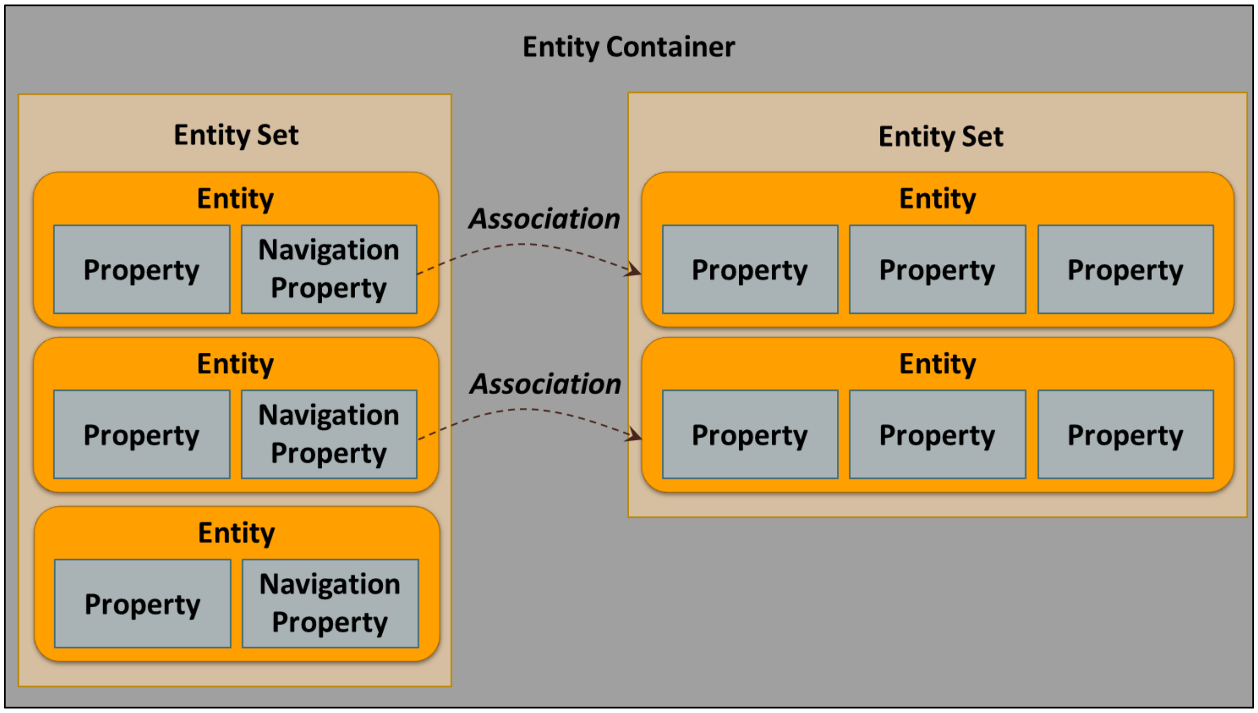

Figure 7: In the EDM, an entity container holds entity sets, while each entity has one or more properties.

As the figure shows, the EDM organizes entities into a simple hierarchy. Each entity is part of an *entity set*, and each entity set belongs to an *entity container*. Entities, each of which is of some entity type, also have a simple structure: They contain *properties*, each of which contains data that this entity holds. To describe the data in properties, the EDM defines a variety of data types, such as String, Boolean, Int16, Int32, Binary, and DateTime.

Special properties called *navigation* properties represent associations—they implement connections between entities. In the example show here, for example, each entity set might be a table in a relational database, with each entity a row in that table. Navigation properties represent relationships between rows, such as those expressed by foreign keys.

Having a general model for all kinds of data is essential. Without it, OData couldn't give clients a common view of diverse data sources. Useful as it is, though, the EDM isn't enough. There must also be a way for an OData client to send requests to an OData service, then get data back. The OData protocol defines how to do this, as described next.

The OData Protocol

The OData protocol is based on REST; at bottom, it's just HTTP. But HTTP alone isn't enough. OData also defines how data modeled using the EDM should look on the wire, how to form queries against that data, and more. This section takes a closer look at these aspects of the technology.

Protocol Basics

An OData client accesses data provided by an OData service using standard HTTP. The OData protocol largely follows the conventions defined by REST, which define how HTTP verbs are used. The most important of these verbs are:

- GET: Reads data from one or more entities.
- PUT: Updates an existing entity, replacing all of its properties.
- MERGE: Updates an existing entity, but replaces only specified properties[2].
- POST: Creates a new entity.
- DELETE: Removes an entity.

As usual with REST, each HTTP request is sent to a specific URI, identifying some point in the target OData service's data model. For example, the root URI for a service might be `www.fabrikam.com/example`.

A client can typically learn about the data model used by an OData service by issuing a GET on a service's root URI with "$metadata" appended to it. For example, issuing the HTTP request

```
http://GET www.fabrikam.com/example/$metadata
```

returns a description of the EDM schema for the data model exposed by this OData service. The returned schema is expressed in an XML-based format called the *conceptual schema definition language (CSDL)*, and an OData client can examine it to see what the service's data model looks like.

Like most data access protocols, OData must handle authentication: How does a client prove its identity to an OData service? The answer is that since OData is based on REST, any authentication scheme that works in a RESTful context will work here. For straightforward interactions, communication between OData clients and services can rely on HTTP Basic Authentication over SSL. For more complex scenarios, Microsoft recommends using OAuth 2.

Serializing Data with Atom/AtomPub

The purpose of the OData protocol is to let a client get data from an OData service. While the EDM defines an abstract data model, it says nothing about how that data should be serialized, i.e., how it should be represented on the wire. To fill this gap, OData today defines two serialization options: one using the XML-based Atom/AtomPub and another using JSON. Both are worth looking at, beginning with the more commonly used Atom/AtomPub.

Atom, defined in RFC 4287, was originally created to describe information in blogs. It models a blog as a *feed* that provides data to its readers. Each feed contains some number of *entries*, each of which holds the content of a

[2] MERGE is a custom HTTP method added by OData's creators. Since then, RFC 5789 has defined the PATCH method to provide the same functionality. The next version of OData will support both MERGE and PATCH.

particular blog entry. AtomPub, officially known as the Atom Publishing Protocol, defines the notion of a *service* that contains one or more *collections*. It also defines a set of RESTful interactions for accessing a service.

Taken together, Atom and AtomPub define a hierarchical model for data, as Figure 8 shows.

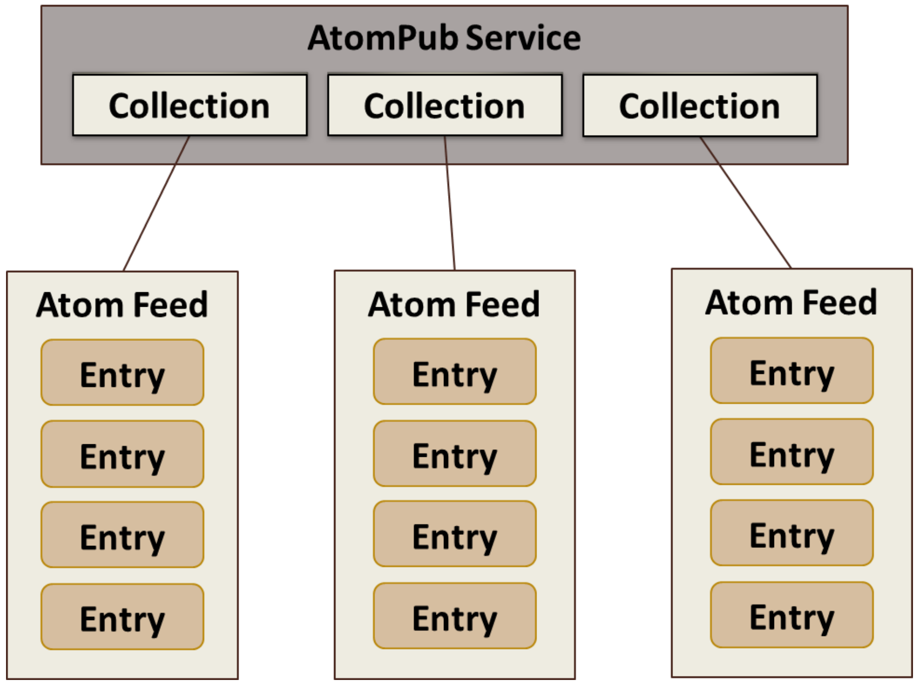

Figure 8: Atom and AtomPub together define an XML representation of data organized into a hierarchy.

In the Atom/AtomPub world, each collection is mapped to a feed. For a client to learn what blogs a particular site makes available, it can ask the AtomPub service for the collections it contains, then access the feed that each collection represents. The Atom and AtomPub specifications define how to represent this in XML, providing a concrete way to send information across the network.

All of this raises an obvious question: What does a data model originally created for blogs have to do with the kind of general purpose data access that OData allows? The answer is that from its humble blog origins, Atom/AtomPub has grown into a widely used approach for working with a variety of data on the Web. (In fact, the creators of AtomPub explicitly intended to design something that would be more broadly useful.) Given this popularity, OData's creators chose to adopt it for an XML serialization format rather than create something new.

Like the EDM, Atom/AtomPub organizes information into a hierarchy: A service contains collections, each of which corresponds to a feed, with each feed containing entries. Mapping the EDM to Atom/AtomPub is straightforward, as Figure 9 shows.

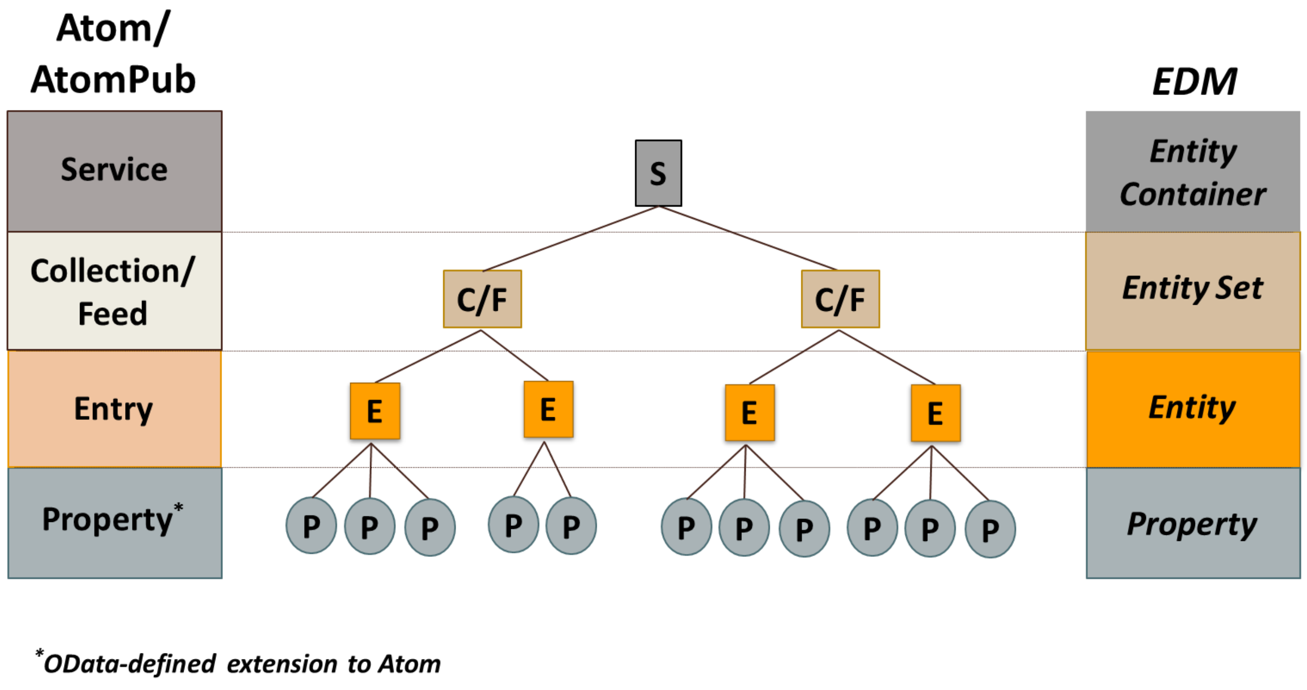

Figure 9: OData can use Atom/AtomPub to serialize EDM-defined data for transmission across the wire.

As the figure shows, an AtomPub service corresponds to an entity container in EDM. An AtomPub collection, together with the Atom feed it's associated with, is mapped to an EDM entity set. An Atom entry corresponds to an EDM entity, while both hierarchies represent actual data values as properties. (Atom doesn't define properties, however—this is an extension added by OData.)

To get a concrete sense of how these abstractions are used, think about how an OData service might map data in a relational database first into the EDM, then into Atom/AtomPub for transmission to a client. Figure 10 summarizes the relationships.

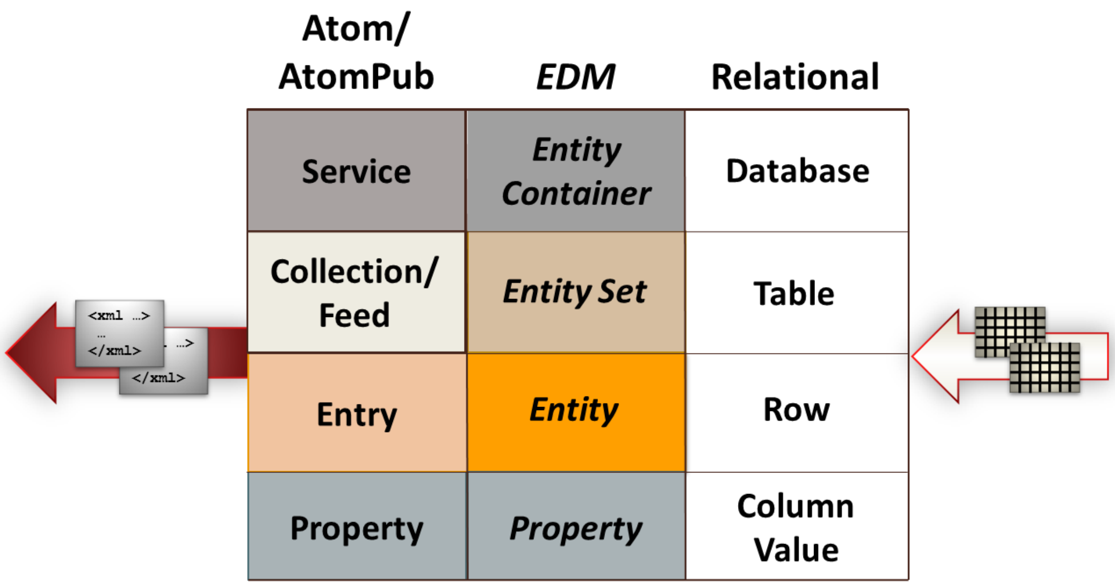

Figure 10: Relational data can be mapped first to the EDM, then to Atom/AtomPub for transmission to an OData client.

Like the EDM and Atom/AtomPub, a relational database organizes data into a hierarchy. At the top is the database itself, which contains tables. Each table holds some number of rows, while each row contains a set of column values. Mapping this to the EDM and Atom/AtomPub, the database itself corresponds to an EDM entity container, then to an AtomPub service. Each table becomes an EDM entity set, represented as an AtomPub collection and an Atom feed. Each row in the table is an EDM entity and an Atom entry, while each column value becomes a property in both EDM and Atom.

To understand how this actually looks on the wire, it's useful to walk through an example. Figure 11 shows a simple relational database with two tables: Customers and Orders. Both tables have three columns, and both are exposed by an OData service with the URI `www.fabrikam.com/example`. To begin accessing the data this service provides, an OData client can issue an HTTP GET on this URI, as Figure 11 shows.

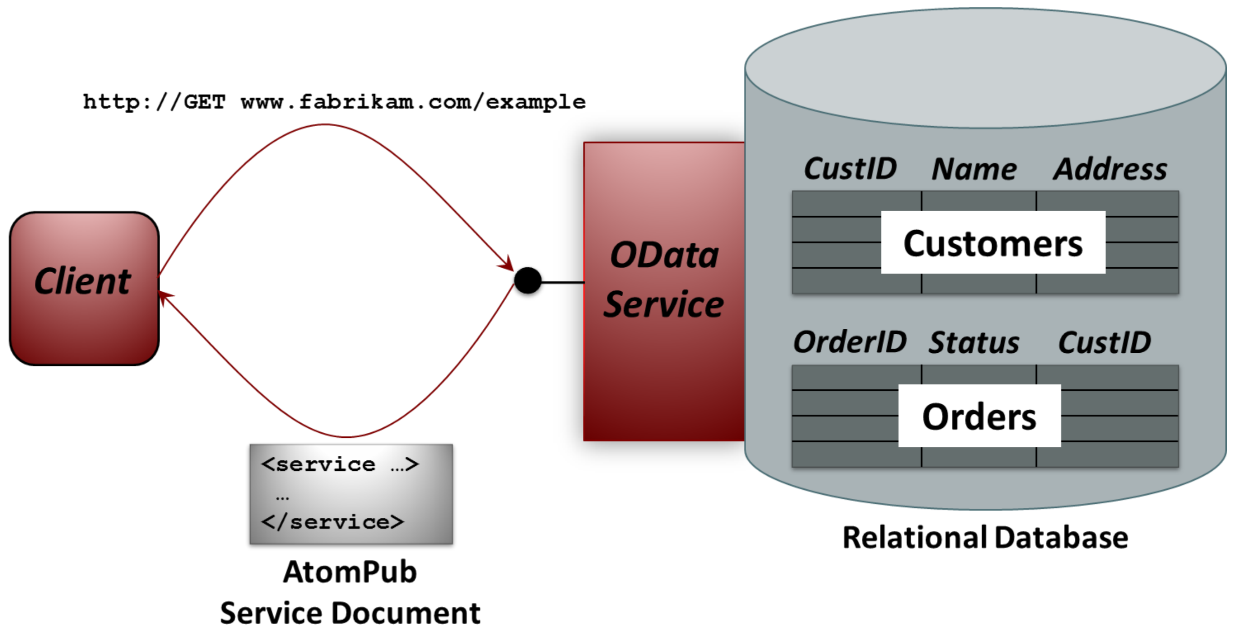

Figure 11: Issuing a GET on the base URI of an OData service returns an AtomPub service document describing the collections (i.e., the feeds) in that service.

The result of this GET is an AtomPub *service document*. As its name suggests, a service document contains a list of what's available in this service, that is, what collections the service contains. Here's a sketch of what that service document looks like for the example in Figure 11:

```
<service …>
…
  <collection href="Customers">
    <atom:title>Customers</atom:title>
  </collection>
  <collection href="Orders">
    <atom:title>Orders</atom:title>
  </collection>
…
</service>
```

As described earlier, each of the database's tables is represented as an AtomPub collection. The client can then access a particular table with another GET request. For example, to access the Orders table, the client issues a GET on the URI `www.fabrikam.com/example/Orders`. Doing this returns an Atom *feed document*, as shown in Figure 12.

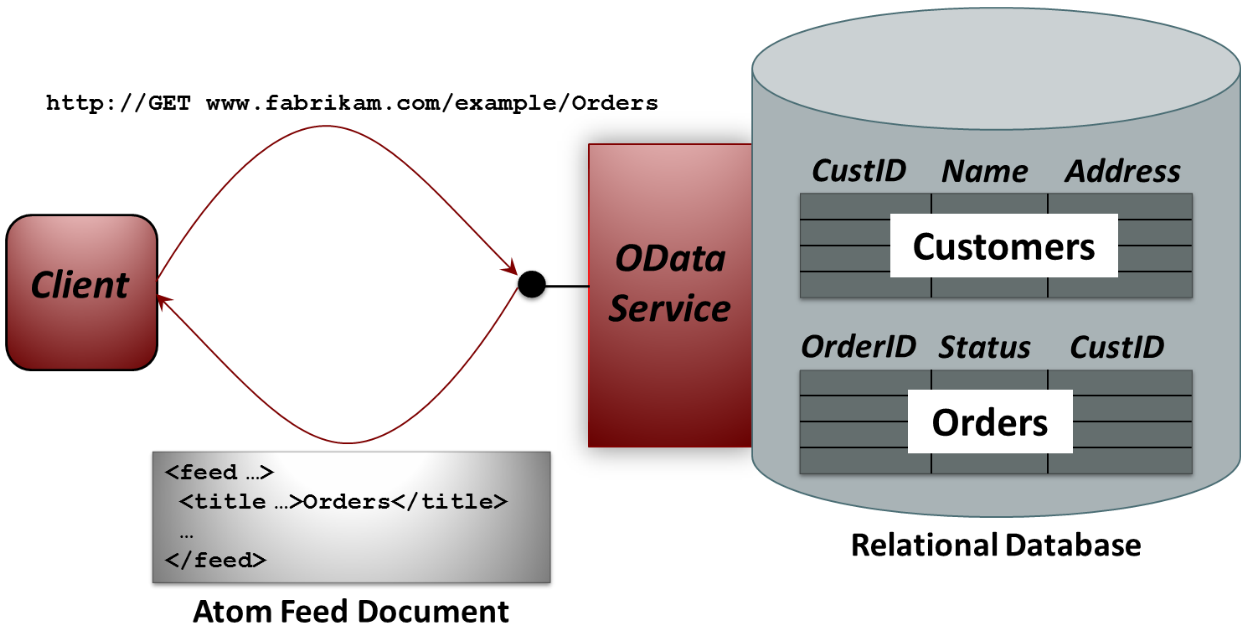

Figure 12: Issuing a GET on a feed's URI returns a feed document containing that feed's data, such as the contents of a table.

Because an EDM entity set (and thus a table) is mapped to a feed, each feed document references a single table. Doing a GET on a feed returns the entire contents of the Orders table in a feed document with each row as an entry. Here's a simplified version of how that document looks:

```
<feed xmlns:m=http://schemas.microsoft.com/ado/2007/08/dataservices/metadata
      xmlns:d=http://schemas.microsoft.com/ado/2007/08/dataservices … >

  <title type="text">Orders</title>
    …
  <entry>
    …
    <content type="application/xml">
      <m:properties>
        <d:OrderID m:type="Edm.Int32">3501</OrderID>
        <d:Status>Shipped</Status>
        <d:CustID m:type="Edm.Int32">867734</CustID>
      </m:properties>
    </content>
  </entry>
  <entry>
    …
    <content type="application/xml">
      <m:properties>
        <d:OrderID m:type="Edm.Int32">5630</OrderID>
        <d:Status>Placed</Status>
        <d:CustID m:type="Edm.Int32">8499734</CustID>
      </m:properties>
    </content>
  </entry>
  <entry>
    …
  </entry>
  …
</feed>
```

The feed document begins by referencing XML namespaces containing types that OData needs. It then specifies the title of the feed (i.e., the table) being returned, followed by an XML element for each entry (i.e., each row) in that table. In this simple example, the table has only three columns, so each entry has three properties. Notice that properties and their data types are borrowed from the EDM—Atom has no notion of either one. And since the default type is `string`, there's no need to explicitly specify a type for the characters in the `Status` property.

It's also possible to request just parts of a table. For example, issuing a GET on the URI `www.fabrikam.com/example/Orders(5630)` returns an Atom feed document containing only the entry whose primary key is 5630. Assuming that the OrderID column is the key for this table, that document would contain just the second of the two entries shown above. Another option is to use the $count option to request just the number of entries in a feed. For example, issuing the request

```
http://GET www.fabrikam.com/example/Orders/$count
```

returns the number of entries (that is, the number of rows) in the Orders table.

EDM data serialized using Atom/AtomPub can also represent associations. Even though EDM models associations with navigation properties, the Atom/AtomPub serialization doesn't use properties for associations. Instead, they're represented using AtomPub `link` elements.

Atom/AtomPub can be used to serialize any set of EDM-defined data—it's not just for relational tables. For example, a group of SharePoint lists can be viewed as an AtomPub service, with each individual list an Atom feed. Each of a list's items then becomes an entry, with each field in that entry represented as a property. Other data sources can be mapped in a similar way.

Serializing Data with JSON

JavaScript Object Notation is a way to represent information on the wire and in other contexts. Described in RFC 4627, JSON is derived from the syntax of JavaScript, which makes life simple for JavaScript developers. Unsurprisingly, JSON has become a popular choice for data sent to applications running in Web browsers. JSON support is available for multiple languages, however—it's not only for JavaScript—and because it provides a simple and compact way to represent data, it's a useful option for serializing information.

Whether an OData client chooses to have the data it receives serialized using Atom/AtomPub or JSON, the way it interacts with an OData service is the same. In both cases, the client can typically ask an OData service for a document that describes all of the available entity sets (collections/feeds in the parlance of Atom/AtomPub), then request the data those entity sets contain.

For example, a client using JSON could ask the example OData service shown earlier for a list of all of its entity sets with the same request used to get an AtomPub service document:

```
http://GET www.fabrikam.com/example
```

With JSON, however, the resulting document looks like this:

```
{ "d" : {
   "EntitySets": ["Customers", "Orders"]
}  }
```

The “d” stands for “data”, and it’s used to wrap all JSON results returned from an OData service. Here those results are the two entity sets Customers and Orders, representing the two relational tables exposed by this service.

A client can then request entities in those entity sets, which in this example means rows in tables. For instance, suppose a client asks for the row in the Orders table with the key 5630 by issuing the request

```
GET http://www.fabrikam.com/example/Orders(5630)
```

The result in JSON looks like this:

```
{ "d" : {
   "results": {
     "OrderID": 5630,
     "Status": "Placed",
     "CustID": 8499734 }
}  }
```

It’s worth re-emphasizing that the approach OData uses to model data—the EDM—is entirely separate from its serialization formats of JSON and Atom/AtomPub. (Even though it’s common to talk about “OData feeds”, for instance, Atom feeds aren’t a required part of the technology.) The separation between the EDM and serialization also means that the OData URIs and protocol semantics are the same regardless of the serialization format used. This separation makes it straightforward to add other serialization formats in the future; the technology isn’t wedded to what’s popular today.

Issuing Queries

Invoking a GET on a URI is useful, but it’s a blunt instrument. Clients also need a way to issue more precise queries on data, something that Atom/AtomPub doesn’t define. To meet this need, OData defines a query language.

The OData data model isn’t relational, and so this query language isn’t SQL. Instead, it defines a set of options that can be appended to a GET request. The possibilities include:

- $top=n: Returns only the first *n* entities in an entity set (or in Atom terms, the first *n* entries in a feed).
- $skip=n: Skips the first *n* entities in an entity set. Using this option lets a client retrieve a series of distinct pages on subsequent requests.
- $format: Determines whether data should be returned in JSON or the XML-based Atom/AtomPub format. (The default is Atom/AtomPub.)

- $orderby=<*expression*>: Orders results, in ascending or descending order, by the value of one or more properties in those results.

- $filter=<*expression*>: Returns only entities that match the specified expression.

- $select=<*expression*>: Returns only the specified properties in an entity.

The last three options all rely on OData expressions. These expressions can contain many different operators, all of which are represented as text. For example, equality is indicated by the string “eq”, greater than by “gt”, and addition by “add”[3]. To return, say, entities for all orders whose OrderID is between 3,000 and 5,000, an OData client could issue this HTTP request:

```
http://GET www.fabrikam.com/example/Orders?$filter=OrderID gt 3000 and
OrderID lt 5000
```

Along with operators, the OData query language provides string operations, date/time operations, and more.

An OData service can also define *service operations*, pre-defined functions that clients can invoke. For example, a service that stores pictures might expose a service operation that lets clients apply various effects to an image. Service operations can also be used to expose stored procedures from a database or in other ways.

A Perspective: OData in a SOA World

For anybody familiar with service-oriented architecture (SOA), OData raises an obvious question: Why not use a SOA approach to exposing data? Why invent something new with OData?

As described earlier, one clear advantage of OData is its commonality: Rather than having each application define its own unique set of services, they can all use the common data model and protocol defined by OData. This makes it easier to mix and match clients and data sources, and it also allows using common libraries to create both.

But there’s also another reason why OData can be better for exposing data than a traditional SOA approach. Suppose an application needs to expose data about customers and orders to various clients. With the SOA approach, the application’s creators would define a set of custom operations (using either REST or SOAP) that clients could call. They might, for example, create a GetCustomer operation that returned data about one or more customers. If the application’s first client wanted the ability to retrieve customers by customer number, this operation might have a parameter that allowed making exactly this kind of request.

Now suppose another client wants to access this application’s data through the same interface. This client needs to retrieve customer data by geographic area, however, rather than by customer number. To allow this, the creator of the application’s interface likely adds a new operation, perhaps called GetCustomerByGeo. When yet another client appears, it might well want to retrieve customer data in some other way, requiring still more changes to the interface.

[3] Using standard symbols such as “=”, “>”, and “+” might seem simpler, but most of these already have a meaning in the URI syntax—they’re not available for OData to use.

It’s common to see this kind of evolution in SOA interfaces today, and it’s not a pretty thing. The problem grows out of a fundamental limitation, however, which is that services expose only specific views into an application’s data. It’s as if the data is behind a curtain, with each service punching a hole in that curtain. The creator of this interface is forced to guess how the application’s clients will want to access the data, deciding in advance what holes to cut in the curtain.

Contrast this with how OData works. Rather than exposing application-defined services, it allows access to an entire application-defined data model. Clients are free to query this model in any way they find useful. The application’s creator still has control over what’s exposed, of course, and designing the data model likely requires making some predictions about what clients will want. Still, exposing a data model pulls back the curtain on an application’s data, giving clients more flexibility in how they access it.

Not every problem that relies on SOA today should use OData, of course. If an application exposes a functionality-oriented set of services rather than services targeting data access, for instance, OData probably isn’t a good choice. But for applications that wish to expose data in a broader, more consistent way to diverse clients, using OData can be an attractive option.

OData Client Libraries

A developer is always free to create an OData client from scratch—it’s just code. To make life simpler, though, Microsoft and others provide a number of OData client libraries. The currently available options include libraries for versions 3.5 and 4 of the .NET Framework, JavaScript, Silverlight, Java, PHP, Android, iOS, Windows Phone 7, and Ruby.

Microsoft also provides some tool support for creating OData clients in Visual Studio. For example, when a developer adds a reference to an OData service, Visual Studio 2010 can read that service’s CSDL definition using the $metadata option described earlier. Once it has this information, Visual Studio will generate C# or Visual Basic classes for working with the data this service provides. The tool can also provide a graphic illustration of a service’s data model, giving the developer a visualization of how data exposed by an OData service is structured.

OData Services

As with clients, creating an OData service from scratch is always possible. Once again, though, relying on an existing library can help. Java developers can use the open source odata4j toolkit to help do this, for example. Among other things, this toolkit provides support for exposing data through a Java Persistence API (JPA) entity model or from plain old Java objects (POJOs).

For .NET developers, Microsoft provides WCF Data Services. Figure 13 shows the main parts of this technology.

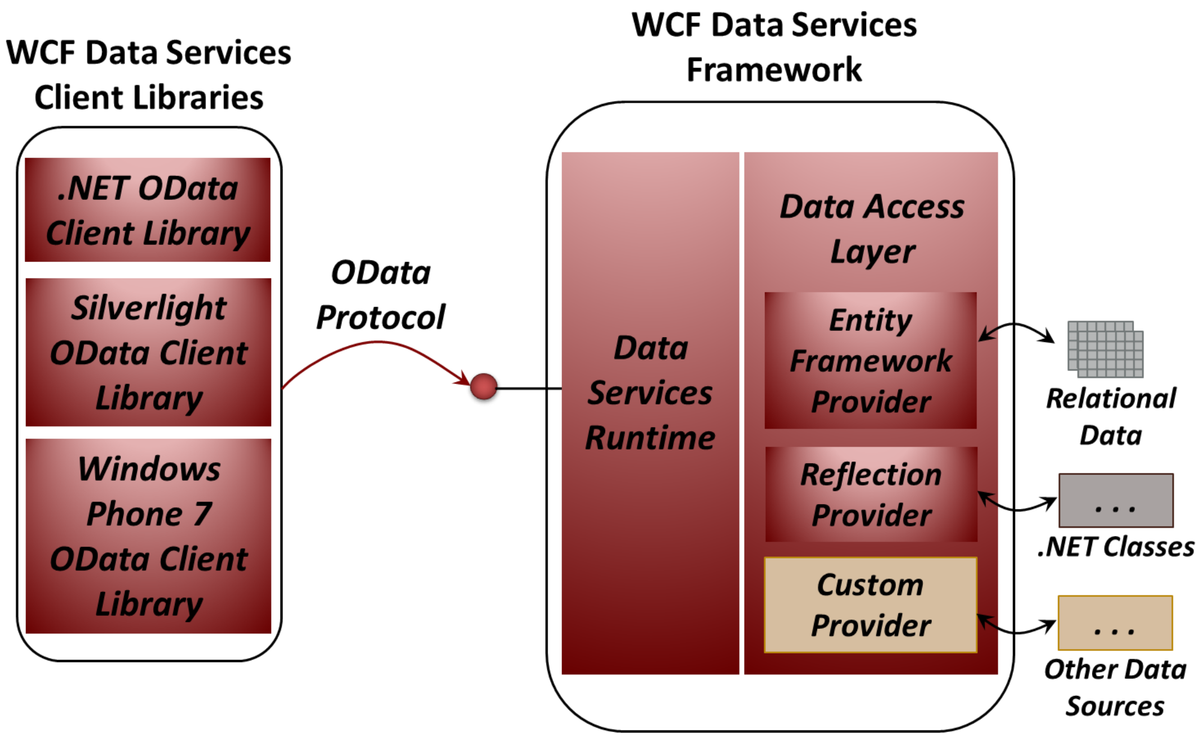
WCF Data Services Client Libraries
.NET OData Client Library
Silverlight OData Client Library
Windows Phone 7 OData Client Library
OData Protocol
WCF Data Services Framework
Data Services Runtime
Data Access Layer
Entity Framework Provider
Reflection Provider
Custom Provider
Relational Data
. . .
.NET Classes
. . .
Other Data Sources

Figure 13: WCF Data Services provides both client and server support for OData.

As the figure shows, WCF Data Services includes the OData client libraries for .NET, Silverlight, and Windows Phone 7. More relevant here, though, is its support for creating an OData service with the WCF Data Services framework. This framework is divided into two parts:

- The Data Services runtime, which implements the OData protocol, parses URIs, and more. This runtime also supports serializing data using both Atom/AtomPub and JSON.
- The Data Access layer, which handles the translation between the underlying data and the EDM. Different kinds of data can use different *providers* to do this. The Entity Framework provider is used to expose relational data, such as data stored in SQL Server. The Reflection provider is used to expose data directly from .NET classes. For other kinds of data, the creator of an OData service can implement a custom provider that does the required mapping.

The more data sources that support OData, the more useful OData becomes. The goal of providing WCF Data Services and odata4j is to make this support easier to create.

Conclusion

Our world really is awash in data. Yet too much of it is locked up in silos, inaccessible to many of the applications that might use it. By providing a common way for diverse clients to access an array of data sources, OData can help set this information free.

Because it relies on a simple abstract data model based on entities and associations, OData can be used with many kinds of data. Because it builds on familiar technologies such as REST, Atom/AtomPub, and JSON, OData isn't especially hard to understand. And because support is available for creating clients and services on various platforms and devices, OData is straightforward to implement.

The value of a common approach to accessing data is undeniable. Reflecting this, many clients and data sources support OData today. Going forward, expect to see that support get broader still. Our data is just too valuable to keep locked away.

For Further Reading

For more detail on OData, go to www.odata.org. The site provides documentation for the protocol (http://www.odata.org/developers/protocols), along with lists of clients (http://www.odata.org/consumers) and data sources (http://www.odata.org/producers) that currently support OData. It also contains other useful information about the protocol and how to use it.

About the Author

David Chappell is Principal of Chappell & Associates (www.davidchappell.com) in San Francisco, California. Through his speaking, writing, and consulting, he helps people around the world understand, use, and make better decisions about new technologies.