Pro ADO.NET Data Services: Working with RESTful Data

Copyright © 2009 by John Shaw and Simon Evans

ISBN-13 (pbk): 978-1-4302-1614-8

ISBN-13 (electronic): 978-1-4302-1615-5

Printed and bound in the United States of America 9 8 7 6 5 4 3 2 1

Distributed to the book trade worldwide by Springer-Verlag New York, Inc., 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax 201-348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `http://www.springeronline.com`.

For information on translations, please contact Apress directly at 2855 Telegraph Avenue, Suite 600, Berkeley, CA 94705. Phone 510-549-5930, fax 510-549-5939, e-mail `info@apress.com`, or visit `http://www.apress.com`.

The source code for this book is available to readers at `http://www.apress.com`.

CHAPTER 1

The Foundations of ADO.NET Data Services

Software development is hard; one year in development is equivalent to a dog year's worth of change. The problem of rapid change is exacerbated by those in the industry who incessantly call the latest innovation the silver bullet that will solve the world's ills. While this is never the case, innovations in technology, if used appropriately, do deliver better solutions that improve the bottom line of the businesses we serve.

Underpinning technology innovations are concepts, which evolve at a much slower rate than the specifics of a new technology. Concepts help us solve many of the requirements that are common to most of the solutions that we as software developers have to deliver in the enterprise.

ADO.NET Data Services is a new technology designed to meet one of the most common requirements in software development: access to data over a network. Developers use three key concepts to meet this requirement, each of which we'll discuss in this chapter:

- The concept of **set-based logic** to access data from a data source
- The concept of **object orientation** to apply business logic to data
- The concept of **service orientation** to communicate data using messages across a network

This chapter will describe the key Microsoft technologies that apply to using these concepts and explore how ADO.NET Data Services builds on top of these foundations to make working with data over a network more productive than ever before.

The Concept of Set-Based Logic

Set-based logic is about solving a problem in terms of sets of data, where a set of data is defined by statements that limit, group, and order the data in the set. For example, you may wish to retrieve a set of customers where the customers' first names equal "Joe" and order the set of customers by their last names. When developers think of set-based logic, they normally think of Structured Query Language (SQL), which is the most common way of implementing set-based logic. A limitation of SQL as a technology is that it is only used to retrieve data from a data source (usually a relational database) such as SQL Server. To give developers a way to carry out set-based operations in code, Microsoft introduced Language Integrated Query (LINQ) in .NET 3.5, which applies the concept of set-based logic to work with all forms of data, whether the information comes from SQL Server or XML, or is in memory.

Under the covers, LINQ uses technologies such as SQL to work with data from the data source, but it abstracts the need for developers to learn several technologies when they have to perform set-based logic. Flavors of LINQ include LINQ to Objects, LINQ to SQL, and LINQ to XML.

The Concept of Object Orientation

Whereas set-based logic is used to access data, object orientation is used to provide a representation of that data to some real-world entities (objects). The reason that object orientation is so important to writing maintainable solutions is that the representation of data as it is stored in a database does not describe the behavior of that information. For example, you could have customers modeled in a `Customer` table in SQL Server, each with a customer type, such as buyer and vendor. Clearly, some ways in which you need to interact with buyers and vendors will be the same, as they are both customers; but in other ways, the business rules you use with buyers may well differ from those customers who are vendors. For example, you may want to provide vendors with access to sales on your web site, but this information must be withheld from buyers because it is sensitive information. Yet both buyers and vendors would need to log in to the web site, so business rules dealing with authentication and authorization would need to apply to all customers.

Object orientation allows both buyer and vendor entities to inherit from a base customer entity, meaning that they can both behave as customers, but also extend this common behavior with buyer- or vendor-specific properties and methods. Figure 1-1 shows this principle, with the `Buyer` and `Vendor` classes inheriting from the `Customer` abstract class. The `Buyer` and `Vendor` classes extend the base class, each adding a property and method associated with how buyers and vendors handle products. Without object orientation, it would not be possible to treat both the `Buyer` and `Vendor` objects as customers.

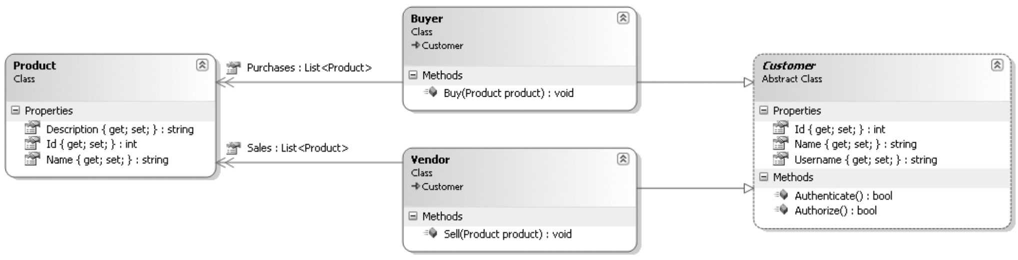

Figure 1-1. *Simple example of the object-oriented principle of inheritance*

In moving between data as it is structured in a relational database and entities, there is clearly a need to map the data between its representation in a table and its representation in an entity. This difference in structure is sometimes referred to as an **impedance mismatch**. Traditionally, this problem would have been solved by writing mapping code in the data access layer, but recent innovations have led to the use of object relational mappers to perform the mapping between databases and objects. By using object relational mapping, the developer does not need to write tedious code to achieve this aim. Several object relational mappers are available to .NET developers, such as NHibernate, but this book will focus on Microsoft's own ADO.NET Entity Framework, because of its close relationship with ADO.NET Data Services and LINQ, using LINQ to Entities.

The Concept of Service Orientation

With set-based logic helping us access data and object orientation helping us implement business logic on that data, the final problem that needs to be solved in order to meet the original requirement is moving this data across a network. The concept of service orientation deals with this requirement by communicating data in discrete messages between a sender and a receiver application. The sender is commonly called the **client** or **proxy** and the receiver is the **service**.

The sent messages must conform to a contract dictated by the service. The contract defines exactly what structure of data is accepted by the service and how the data can be used. The format (such as XML) and encoding (such as text) of the message may differ, and the low-level protocol (transport) used for communication may differ also; however, by considering communication of data as encoded messages being sent and received over a transport using a set contract, service orientation loosely couples the sender and receiver, and acknowledges that the technology implementation of the transport and encoding may change over time. Indeed, we may also want to use different transports and encodings for different clients. For example, we may want to expose a service to external customers over the Internet using the Hypertext Transfer Protocol (HTTP) transport with a text encoding, but internally we may want to communicate using TCP/IP and a binary encoding, because internally we have control of the development of our own clients and the security of our network.

In .NET, the concept of service orientation is delivered using Windows Communication Foundation (WCF), and ADO.NET Data Services is built on top of this foundation. WCF is described in detail later on in this chapter in the section "The ABCs of Windows Communication Foundation."

RESTful Thinking

If you have designed components or services in the past, you are most likely to have developed an interface or service contract, where you define what operations a component is allowed to perform. For example, you might design a contract that contains three operations: `GetProductById`, `PersistProduct`, and `DeleteProduct`. Each of these operations deals with a product entity and defines what you can do with a product. In other words, each operation is a **verb**, and it deals with the product entity, which is a **noun**.

Why do we write service contracts containing operations describing what a client can do with the service? It is because there is commonly no other mechanism for understanding how a service can be consumed. But the majority of **web** services use HTTP as their transport. This protocol specifies HTTP verbs that must be sent in the request message. Therefore, if we are only exposing our service over HTTP, it is possible to use the protocol itself to understand the intent. This is the architecture proposed by Roy Fielding when he first described a style of service-oriented architecture (SOA) known as representational state transfer (REST).

REST services do not require you to design your own service contracts, because you can rely on HTTP verbs to describe intent. This radically simplifies your service's design. HTTP defines the verbs GET, PUT, POST, and DELETE. By using these built-in verbs, REST services provide access to resources (our nouns) via a Uniform Resource Indicator (URI) and an HTTP verb. For example, you could get a product by its ID by specifying `http://myservice.com/Products(1)` using an HTTP GET verb.

By focusing on resources rather than intent, REST services expose a behavior that is unique to this style of SOA; every resource exposed by a service is accessible to the client if the client decides to address it. This open architecture for services is much more akin to how the world uses the Internet as a whole: a pool of resources where consumers decide what they want to access.

The ABCs of Windows Communication Foundation

Anyone who has built software on any scale is likely to have had to work with APIs that deal with communicating information between two or more computers. All forms of communication involve creating messages that are transported over the wire from one computer to another. If you look at this concept from a far enough distance, all applications have similar communication requirements, and yet up until recently developers have had to learn different APIs for different forms of communication. For example, a developer using .NET 1.1 would use ASMX web services (`System.Web.Services`) for communication over HTTP using SOAP messages, the `System.Messaging` namespace for communications over TCP/IP or RPC using MSMQ messages, and remoting (`System.Remoting`) for communications over TCP/IP using binary messages to remoting applications.

While this proliferation of APIs makes the developer's life much harder, meaning they have to learn and relearn a list of APIs that keeps growing as the communication protocols evolve, it has a more damaging impact on the investment made by the owner of your software; as your business changes, the requirements you have of your software inevitably change, which can include changing how your solutions need to communicate with other solutions. This inevitably means switching communication APIs and throwing away much of your existing investment. Furthermore, if you want to extend your application to support more than one form of communication, you will have to duplicate effort in your architecture by supporting more than one API.

In.NET 3.0, Microsoft introduced WCF to provide a universal API for communications to and from the Windows platform. This rich framework is able to serve as such an API because it abstracts all the common behaviors that communication protocols share into its architecture, and provides extensibility points to accommodate future changes and additions to communication protocols.

You may well be wondering why such an undertaking has not been conducted before. While the concept of communications via messages is simple to grasp, the intricate differences between each communication protocol make the job of abstraction a monumental task. WCF was over two years in the making before the first version was released. When this work was started back in 2003, many in the industry thought that all web services would adopt SOAP over HTTP as the standard communications protocol for web services.

WHAT IS SOAP?

SOAP, which stands for Simple Object Access Protocol, is a W3C messaging specification based on XML. The specification defines three major parts to a message structure: the envelope, the header, and the body.

The SOAP header and the body are both contained within the envelope, which is the root element of a SOAP message. The header contains important metadata that relates to how the message should be used by a service. For example, one common header is the Action header used to address a SOAP message. The body contains the core information that is consumed by a service.

However, in recent years, REST has become more ubiquitous than SOAP as a standard for public services available across the Internet. Fortunately, the WCF team's investment in abstracting the concepts of communication paid dividends, as they were easily able to accommodate the communication requirements of REST in .NET 3.5.

WCF abstracts the communication of messages into three key pillars: address, binding, and contract (ABC). An **address** identifies **who** the intended recipient for the message is. In the Internet world, this is most commonly a URI, but in other communication protocols, the address may take on other forms. A **binding** describes **how** the message will be communicated, including which transport and encoding to use, as well as any other policies such as security that apply to how the message is communicated. For example, a SOAP-formatted message may be sent over an HTTP transport using text encoding. A **contract** describes **what** format the message(s) will take, what a service can do, and what format any faults in the service will take.

Addresses

For anyone who uses the Internet, the concept of an address is most easily associated with entering a Uniform Resource Locator (URL) into a browser to retrieve a web page. The URL points to the location of a resource on the Internet. The URL is a subset of a URI, which defines both the location and the Uniform Resource Name (URN) for the resource. This URI is the address of the web page, which is coupled to the transport you are using (HTTP). REST services such as those you develop using ADO.NET Data Services are entirely reliant on the addressing capabilities of HTTP; once you strip away the semantics of a message sent using HTTP, the content of the message does not include the address of the intended recipient.

While addressing for REST services is a simple matter of understanding that the URI is the address, it is worth understanding the importance of addressing within the broader sphere of messaging architectures and message topologies. REST-based services are broadly used by consumers that adopt a request-reply message exchange pattern; this means the client (such as a browser) sends a request message to a URI address and waits for a response from that URI. However, other important message exchange patterns exist, such as one-way (datagram) messaging, where the client sends a message without waiting for a reply. One-way messaging is an important pattern because it allows high-scale messaging for circumstances where the client need not wait for a response from the service.

A message topology often used with one-way messaging is a **service intermediary**. Consider a service intermediary to be a similar concept to a postman, who delivers your mail without knowing or caring about the content. For example, a service intermediary may be used to take a text-encoded message sent via HTTP over the Internet and forward the message on using binary encoding over TCP/IP. Service intermediaries are only concerned with whom the recipient of the message is, and thus they must understand the address of the message. Because REST services rely on the URI for addressing, it is assumed that this URI is the ultimate recipient of the message, because the message does not contain any other address. Therefore, intermediaries cannot be easily employed using solely REST-based architectures.

■**Note** Another important message topology that relies heavily on the concept of addressing is the **service broker topology**, where the broker acts as a hub to messages published to it and then broadcasts messages to subscribers of the published message. An important example of a service broker is BizTalk Server.

Bindings

In WCF a binding describes how messages are sent and received. The binding is used to create a **channel stack**, where each channel in the stack is a facet of the overall method of communication. A channel stack must always include an encoding and a transport (such as text sent over HTTP), but it can also include other information, such as security or WS-* policies as described by the binding.

WHAT IS WS-*?

WS-* is a set of standardized web service policies for SOAP-based messages. The policies have a number of contributors including Microsoft, IBM, and Sun Microsystems, which ensure their adoption across platforms.

WS-* policies handle common messaging requirements that are not standardized by the SOAP protocol itself. Such policies include WS-Addressing, which standardizes the method used to address a message, WS-Security, which applies message-level security, and WS-ReliableMessaging, which can be used to guarantee message ordering between sender and receiver.

In WCF you can create a binding from scratch, but several commonly used bindings come out the box, including `WebHttpBinding` (since .NET 3.5), which is used by ADO.NET Data Services to describe how RESTful messages are sent and received. Bindings can be configured either in code or, more commonly, within a configuration file. Listing 1-1 shows example bindings for both a `WebHttpBinding` (used for REST services) and `WsHttpBinding` (used by SOAP services that use WS-* policies).

Listing 1-1. *Example WCF Configurations for a REST and WS-* Service*

```
<configuration>
  <system.serviceModel>
    <bindings>
      <wsHttpBinding>
        <binding name="SampleWsHttpBinding">
          <reliableSession enabled="true" />
          <security mode="None">
            <message clientCredentialType="None" negotiateServiceCredential="false"
               establishSecurityContext="false" />
          </security>
        </binding>
      </wsHttpBinding>
    </bindings>
    <services>
      <service behaviorConfiguration="Apress.Services.SampleWebServiceBehavior"
        name="Apress.Services.SampleWebService">
        <endpoint behaviorConfiguration="Apress.Services.SampleWebEndpointBehavior"
          binding="webHttpBinding" name="WebEndpoint"
bindingNamespace="http://schemas.apress.com/"
          contract="Apress.Services.ISampleWebService" />
      </service>
      <service behaviorConfiguration="Apress.Services.SampleWsServiceBehavior"
        name="Apress.Services.SampleWsService">
        <endpoint binding="wsHttpBinding" bindingConfiguration="SampleWsHttpBinding"
          name="WsEndpoint" contract="Apress.Services.ISampleWsService" />
        <host>
          <baseAddresses>
            <add baseAddress="http://localhost/SampleWsService" />
          </baseAddresses>
        </host>
      </service>
    </services>
    <serviceHostingEnvironment aspNetCompatibilityEnabled="true"/>
    <behaviors>
      <endpointBehaviors>
        <behavior name="Apress.Services.SampleWebEndpointBehavior">
          <enableWebScript/>
        </behavior>
      </endpointBehaviors>
      <serviceBehaviors>
        <behavior name="Apress.Services.SampleWebServiceBehavior">
          <serviceMetadata httpGetEnabled="true"/>
          <serviceDebug includeExceptionDetailInFaults="true"/>
        </behavior>
        <behavior name="Apress.Services.SampleWsServiceBehavior">
          <serviceMetadata httpGetEnabled="true"/>
          <serviceSecurityAudit auditLogLocation="Application"/>
        </behavior>
      </serviceBehaviors>
    </behaviors>
  </system.serviceModel>
</configuration>
```

`WebHttpBinding` defines a channel stack with HTTP/HTTPS as its transport and text as its encoding. It differs from other HTTP-based bindings such as `BasicHttpBinding` in that it does not assume the semantics of a SOAP message. Therefore, it does not wrap the contents of the message inside an envelope, and it does not set the address of the message using a URI defined in an action header. Instead, it relies on the URI used by the HTTP protocol itself to address the message.

An encoding is a core part of all channel stacks created by WCF for any binding. It is important to understand that an encoding is not the same concept as serialization. **Serialization** is the process of taking an in-memory object and writing the contents of the object into a message. During object serialization, the serializer will format the message in a particular way, such as in XML or JSON format. Once the message has been serialized, an encoding is then required to specify how the message will be streamed over the wire. For example, an object may be serialized to XML and then encoded into a binary message sent over the wire. In the context of ADO.NET Data Services, all data is transmitted using a text encoding and is serialized either into JSON or Atom format. This is controlled by specifying an Accept HTTP header on the request message.

Contracts

Contracts describe the messages that are exchanged between a client and a service. WCF enables you to design contracts that define the types of data passed using data contracts and how this data can be used through service contracts, which contain one or more operations describing what methods a service supports. An easy way to consider service contracts is that they are the verbs that describe what a service does, and data contracts are the nouns that describe the objects passed between services.

In SOAP services, data contracts and service contracts are represented by XML Schema Definitions (XSDs) that you define. The service exposes a Web Service Definition Language (WSDL) that a client can examine to understand how to consume these contracts.

REST services differ from SOAP services in that they use the HTTP verbs and the URI to describe the service operations (what a service can do). Therefore, REST services have no need for a WSDL to describe service operations. REST services use the verbs of HTTP to describe the action of the message in terms of Create, Read, Update, and Delete (CRUD) semantics. So, for example, if you are reading data from a REST service, the message will use the HTTP GET verb along with the URI called to identify what information to read from the service. Alternatively, if you are updating data from a REST service, you will use the HTTP POST verb and post the data to the service in the request message.

When messages are received by a service's channel stack, the message needs to be passed on to the correct service operation as defined in the service's contract. WCF conducts this wiring job in a class called the `ChannelDispatcher`. In WCF services that use `WebHttpBinding`, the `ChannelDispatcher` handles wiring a URI and HTTP verb to a service operation in a service contract by examining the `WebGet` (for HTTP GET) and `WebInvoke` (for other HTTP verbs) attributes of the operations in a service contract. WCF also provides a class named `UriTemplate`, which can be used to understand URI patterns, defining the format of arguments passed into the service operation via the URI. This is in contrast to SOAP messages that use `BasicHttpBinding`, where the `ChannelDispatcher` examines the message's Action header to determine which service operation to execute.

Endpoint = Address + Binding + Contract

By using the constructs of address, binding, and contract, WCF delivers a conceptual model for building systems that need to communicate using messages. These three collectively describe an endpoint for a service. A service may expose many endpoints, either because the service exposes many contracts or because the service exposes the same contract using different bindings. For example, a service may want to expose a contract using one endpoint configured to use a `WsHttpBinding` for clients consuming the service from the Internet, and a different endpoint exposing the same contract using `NetTcpBinding` for internal clients, where security requirements are different and internal clients are able to take advantage of binary encoding across the LAN.

■**Note** WCF comes with a configuration editor in the SDK to help you safely edit the bindings you use to expose your service endpoints.

Hosting Endpoints

For a service to expose one or more endpoints to clients, the service needs a host in which to control instances of each endpoint at runtime. WCF defines a `ServiceHost` class that reads the description of the endpoint's binding, constructs a channel stack, and opens and closes communications for the service at a given address. In WCF, a host can be any AppDomain, from a console application to a web site running in IIS. IIS is a common choice to host HTTP-based bindings, because it manages process recycling and failure. In Windows Server 2008, Windows Activation Services (WAS) provides similar hosting semantics for non-HTTP-based endpoints. Alternatively, you can host a service outside of IIS or WAS, but you are responsible for keeping that process running and restarting the process after failure.

■**Note** During testing, it is quite common to use a console application as a host for ease of debugging, but in production, services are commonly hosted in IIS or WAS.

In addition to requiring a service host, WCF also defines a factory class called `ServiceHostFactory`, which is used by WCF in managed environments such as IIS to instantiate the `ServiceHost` class from a declarative statement placed in a service host file (an SVC file). You can subclass both the `ServiceHost` and `ServiceHostFactory` classes to add code specific to your hosting requirements. Data services have a predefined service host called `DataServiceHost` and a factory called `DataServiceHostFactory` that together contain all the extra ADO.NET Data Services–specific wiring for hosts of data services. ADO.NET Data Services assumes that you will host your service in IIS (or Visual Studio Web Development Server), which is a fair assumption given that ADO.NET Data Services is tightly coupled to the HTTP transport. Therefore a data service is called by addressing a SVC host file.

Don't Forget the Client

When considering service design, it is equally as important to consider the design of the service from the perspective of a calling client; if your service's contracts are difficult to consume, you will dissuade developers from using your service. Regardless of how you design a service, a client needs to understand how to send and receive messages to and from a service in the correct format over the correct transport to the correct address.

Most WCF clients address this requirement by creating a set of proxy classes, which can be handwritten or generated using `svcutil.exe` (a tool that comes with the SDK). These proxy classes serialize and deserialize to the correct message format and deal with creating the correct channel stack in order to communicate with the service.

Putting It All Together

A summary of all the preceding discussion on WFC leaves us with a generic WCF architecture that looks the one in Figure 1-2.

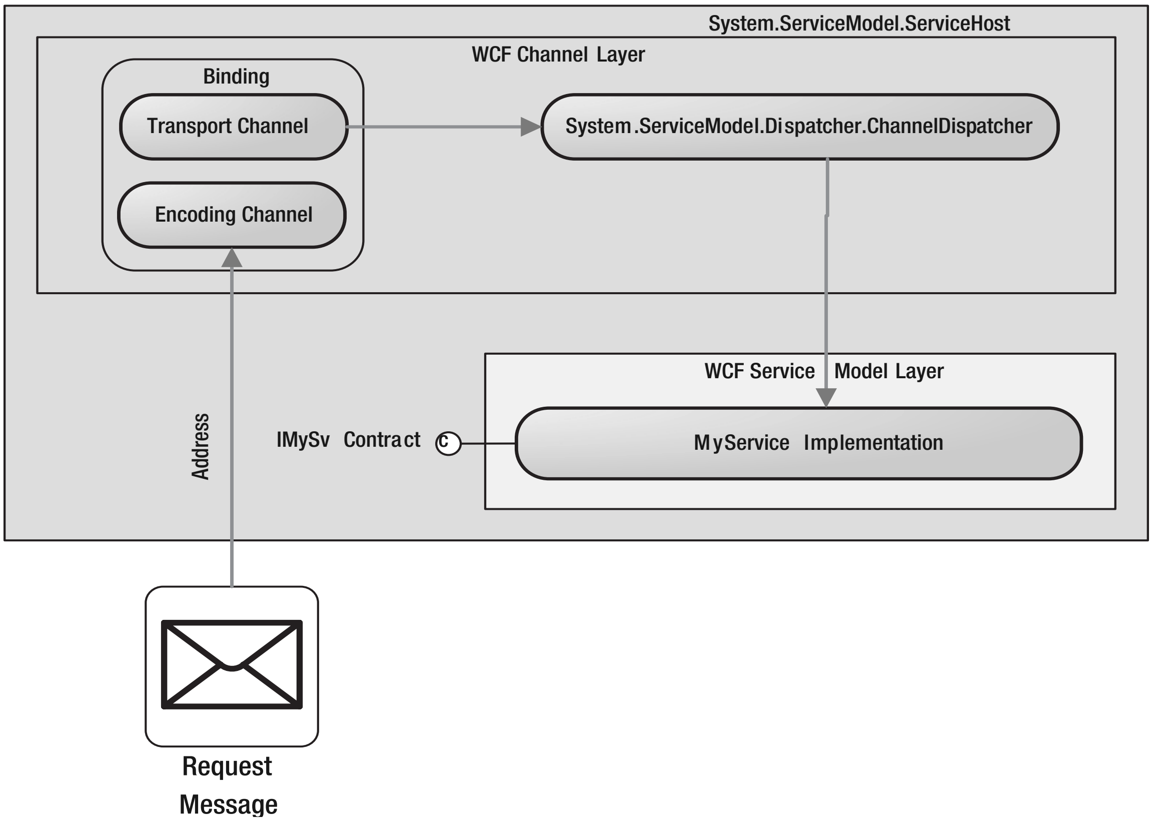

Figure 1-2. *Overview of a generic WCF architecture*

Understanding how WCF defines address, binding, and contract helps you to understand how ADO.NET Data Services has built on top of this foundation, and how this can fit into your broader enterprise SOA.

Standing on the Shoulders of Giants

So what exactly is ADO.NET Data Services? It is best described as the marriage of the technologies that together meet the requirement of accessing data over a network. ADO.NET Data Services uses WCF's ability to create REST-style services, coupled with LINQ's ability to perform set-based logic on a data source such as the Entity Framework, which maps data between a relational database and entities.

So ADO.NET Data Services is standing on the shoulders of giants in the .NET universe. What it adds to this already rich technology stack is simplicity. While WCF, LINQ, and the Entity Framework could be used without ADO.NET Data Services to meet our requirement, ADO.NET Data Services adds an extra layer to the architecture that simplifies the marriage of these technologies. ADO.NET Data Services makes several assumptions about exactly how the requirement of remote data access will be delivered using REST-based services that

- **Only** work using HTTP/HTTPS
- **Only** work using Atom or JSON format messages using a text encoding
- **Only** work using a REST-based URI structure that is dictated for you
- **Only** work using operations that return the LINQ interface `IQueryable`

On the face of it, this may seem very limiting, but in reality most of these limitations are intrinsic to REST services themselves. REST services assume that you are designing services for the Web (and are thus reliant on HTTP as the underlying protocol). While this clearly doesn't apply to all services, it does to the vast majority, and for all other services you can still leverage WCF and build an architecture sympathetic to both needs.

Data services are composed of REST services, and these services use the URI for addressing and calling a service operation, together with a corresponding HTTP verb (such as GET). They use `WebHttpBinding`, which defines text-encoded messages transported over HTTP or HTTPS without any form of message envelope. These services have a predefined format for how to pass arguments via a service URI (similar in concept to a prewritten `UriTemplate`).

The Universal Service Contract

It is well known in software development that the cost of changing code increases dramatically once a version of the software has gone into production. Although change is something that we must all embrace, we should also be aware of any reasonable measures we can take while designing a solution that will limit this cost.

The most costly aspect to developing and evolving services is undoubtedly the cost of changing contracts in production. One of the reasons for such a cost to changing contracts is that you may need to support clients that consume an older version of the contract, while you roll out your new version of the service.

With SOAP services you define what you **can** do with a service; in ADO.NET Data Services you define what you **cannot** do with a service by locking down resources you do not want a client to access and making all accessible resources queryable and addressable. The impact of this REST-based architecture on contract design is that you do not need to specify lots of service operations that define what your service does, and locking down service resources does not involve any contract changes. Thus this architecture reduces the chances of changing your service contracts, because your contracts will only be affected by changes to the resources themselves.

ADO.NET Data Services defines a service contract out of the box named `IRequestHandler`. This contract defines only one service operation called `ProcessRequestForMessage` and is implemented by a class called `DataService<T>`, where `T` is a LINQ data source that implements `IQueryable<T>`. The service operation `ProcessRequestForMessage` enables you to access any information from the underlying data source via a URI and HTTP verb. Therefore, the URI becomes programmable using set-based logic, much like SQL. You define in `DataService<T>` what is not accessible by setting up access rules in the `InitializeService` method.

Atom and JSON

ADO.NET Data Services serializes messages into either Atom or JSON format, depending on the Accept HTTP header in the request message. Atom is a subset of XML, which is defined by a prewritten XML schema. JSON is not XML based, but it is defined by a standard contained in RFC 4627. Both of these message formats have one thing in common, which is that every data item in a **message** contains both the data and the type of data being sent or received. This is in contrast to SOAP services, where the types are defined by a separate XML schema rather than being part of the message itself.

The impact of using either Atom or JSON for serialization is that so long as your client understands the serialization format, it will be able to understand the types of data in the message without needing to refer to a separate contract. Additionally, ADO.NET Data Services makes all related data easily addressable, by exposing URIs for any related resources not returned in the response. Listing 1-2 shows an ADO.NET Data Services raw-text response serialized into Atom format, while Listing 1-3 shows the equivalent data in JSON format.

Listing 1-2. *Example ADO.NET Data Service Response Using Atom format*

```
<?xml version="1.0" encoding="utf-8" standalone="yes"?>
<feed
xml:base="http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/" xmlns:d="http://schemas.microsoft.com/ado/2007/08/
dataservices" xmlns:m="http://schemas.microsoft.com/ado/2007/08/
dataservices/metadata" xmlns="http://www.w3.org/2005/Atom">
  <title type="text">Customers</title>
  <id>http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers</id>
  <updated>2008-09-07T20:00:41Z</updated>
  <link rel="self" title="Customers" href="Customers" />
  <entry>
    <id>http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(3)</id>
    <title type="text"></title>
    <updated>2008-09-07T20:00:41Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Customer" href="Customers(3)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Gender"
type="application/atom+xml;type=entry" title="Gender" href="Customers(3)/Gender" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/
Salutation" type="application/atom+xml;type=entry" title="Salutation"
href="Customers(3)/Salutation" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/
related/Address" type="application/atom+xml;type=feed" title="Address"
href="Customers(3)/Address" />
    <link
rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/TelephoneNumber"
type="application/atom+xml;type=feed" title="TelephoneNumber"
href="Customers(3)/TelephoneNumber" />
    <category term="CustomerModel.Customer"
scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
        <d:CustomerId m:type="Edm.Int32">3</d:CustomerId>
        <d:FirstName>Jane</d:FirstName>
        <d:LastName>Smith</d:LastName>
        <d:DateOfBirth m:type="Edm.DateTime">1982-03-01T00:00:00</d:DateOfBirth>
      </m:properties>
    </content>
  </entry>
  <entry>
    <id>http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(5)</id>
    <title type="text"></title>
    <updated>2008-09-07T20:00:41Z</updated>
    <author>
      <name />
    </author>
    <link rel="edit" title="Customer" href="Customers(5)" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/Gender"
type="application/atom+xml;type=entry" title="Gender" href="Customers(5)/Gender" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/
Salutation" type="application/atom+xml;type=entry" title="Salutation"
href="Customers(5)/Salutation" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/
Address" type="application/atom+xml;type=feed" title="Address"
href="Customers(5)/Address" />
    <link rel="http://schemas.microsoft.com/ado/2007/08/dataservices/related/
TelephoneNumber" type="application/atom+xml;type=feed" title="TelephoneNumber"
href="Customers(5)/TelephoneNumber" />
    <category term="CustomerModel.Customer"
scheme="http://schemas.microsoft.com/ado/2007/08/dataservices/scheme" />
    <content type="application/xml">
      <m:properties>
```

```
        <d:CustomerId m:type="Edm.Int32">5</d:CustomerId>
        <d:FirstName>John</d:FirstName>
        <d:LastName>Smith</d:LastName>
        <d:DateOfBirth m:type=”Edm.DateTime”>1963-09-09T00:00:00</d:DateOfBirth>
      </m:properties>
    </content>
  </entry>
</feed>
```

Listing 1-3. *Example ADO.NET Data Service Response Using JSON Format*

```
{ "d" : [
{
"__metadata": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(3)", "type": "CustomerModel.Customer"
}, "CustomerId": 3, "FirstName": "Jane", "LastName": "Smith", "DateOfBirth":
"\/Date(383788800000)\/", "Gender": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(3)/Gender"
}
}, "Salutation": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(3)/Salutation"
}
}, "Address": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(3)/Address"
}
}, "TelephoneNumber": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(3)/TelephoneNumber"
}
}
}, {
"__metadata": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(5)", "type": "CustomerModel.Customer"
}, "CustomerId": 5, "FirstName": "John", "LastName": "Smith",
"DateOfBirth": “\/Date(-199238400000)\/”, "Gender": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(5)/Gender"
}
}, "Salutation": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(5)/Salutation"
}
}, "Address": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(5)/Address"
}
}, "TelephoneNumber": {
"__deferred": {
"uri": "http://localhost.:1478/Apress.Data.Services.CustomerService.Host/
CustomerDataService.svc/Customers(5)/TelephoneNumber"
}
}
}
] }
```

A data service is able to understand the types of data from the underlying data source and emit these types within messages serialized into Atom or JSON format. This means that there is no need to develop your own data contracts as you would have done with SOAP services because the serialization format and the data source define these contracts for you.

■ **Note** You should be aware however, that making changes to your underlying data source (such as the Entity Framework) will still effectively force you to version your service for customers who are still consuming the older data model.

The reason ADO.NET Data Services supports both Atom and JSON formats is that they are both useful in different circumstances. Atom-formatted messages are useful in cases where the client can work most effectively with XML (such as a Silverlight 2.0 client). The JSON format serializes messages into an object array format native to JavaScript clients (such as an HTML web page running in a browser). Browsers are therefore able to work with JSON-formatted data more efficiently than XML-formatted data because the former can be manipulated natively in JavaScript.

Data Service Hosting

ADO.NET Data Services assumes that your service will be hosted in IIS (or Visual Studio Web Development Server) and provide a `DataServiceHost<T>`, which supplies that additional wiring for building an ADO.NET Data Services host.

Data Service Clients

ADO.NET Data Services clients need to concern themselves with amending the HTTP `Accept` header in their HTTP request to ensure that a response is returned using the required data format (Atom or JSON). Additionally, with REST services such as those built using ADO.NET Data Services, it is not essential to create a set of proxy classes, because the address and transport are defined by HTTP. As most clients understand how to communicate over HTTP using a URI, it is possible to just create a simple HTTP request and parse the response. For example, you could use the `WebClient` class in the .NET Framework to create a simple HTTP request using HTTP GET.

Whether or not you use a set of proxy classes will depend on the type of client you are developing; a client developed for Ajax use (using JSON-formatted messages) will send a request using the browser's XMLHTTP capabilities. In this case, communications will be instigated by JavaScript running in a browser. Frameworks such as ASP.NET AJAX simplify the consumption of data services in JavaScript by providing prebuilt JavaScript libraries that deal with issues such as those relating to browser compatibility.

If you are consuming a data service from .NET code, you can use a tool provided with the ADO.NET Data Services SDK named `DataSvcUtil.exe`, which generates a .NET proxy for your service. The additional benefit this proxy has over just creating an HTTP request is that it understands how to serialize and deserialize Atom, and also provides LINQ to Data Services, which enables you to query your data service as though it were any other LINQ-based data source.

Putting It All Together

A summary of all the discussion of how ADO.NET Data Services fits in WCF leaves us with an ADO.NET Data Services–WCF architecture that looks like Figure 1-3.

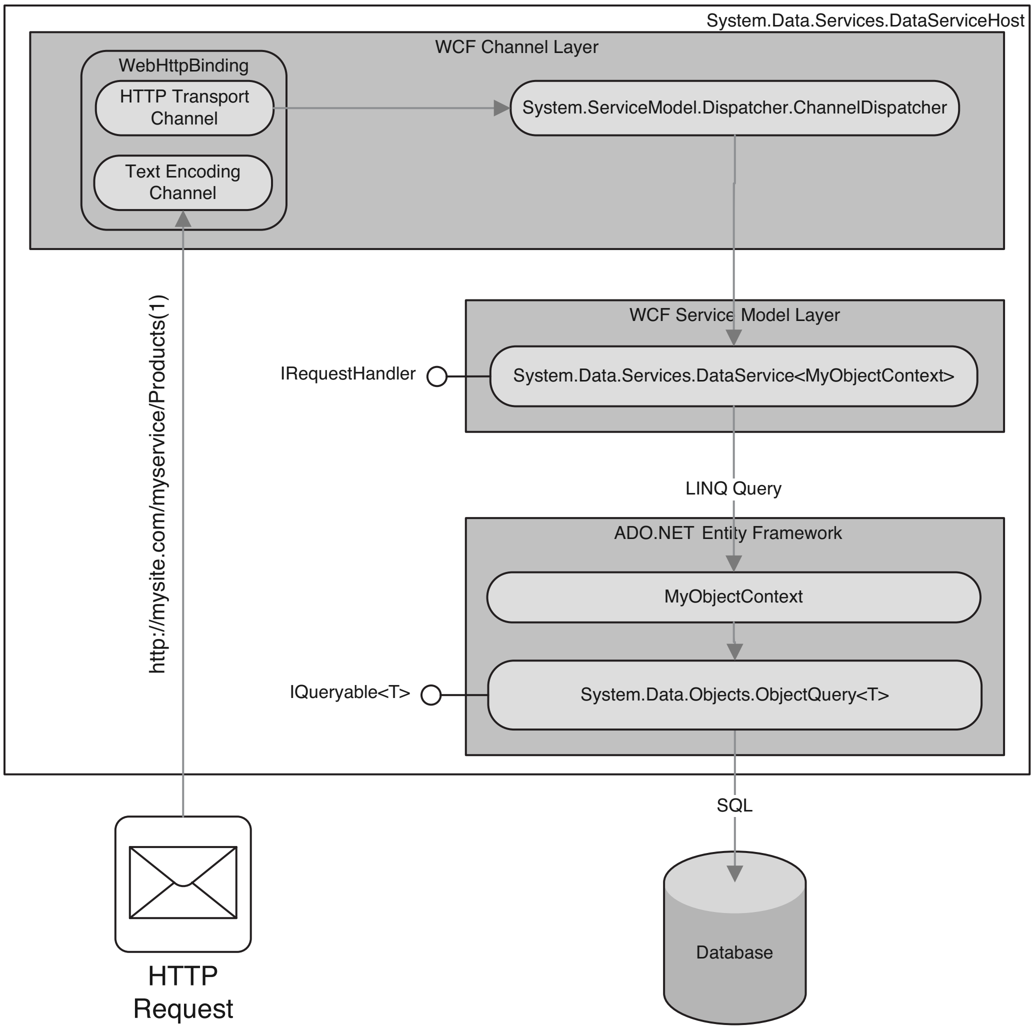
System.Data.Services.DataServiceHost
WCF Channel Layer
WebHttpBinding
HTTP Transport Channel
Text Encoding Channel
System.ServiceModel.Dispatcher.ChannelDispatcher
WCF Service Model Layer
IRequestHandler
System.Data.Services.DataService<MyObjectContext>
LINQ Query
ADO.NET Entity Framework
MyObjectContext
IQueryable<T>
System.Data.Objects.ObjectQuery<T>
http://mysite.com/myservice/Products(1)
SQL
HTTP Request
Database

Figure 1-3. *Overview of the ADO.NET Data Services architecture*

From the preceding figure you can see how ADO.NET Data Services fits into broader technology stack of WCF and LINQ.

Reports of My Death Have Been Greatly Exaggerated

Looking at the architecture of REST services and how ADO.NET Data Services implements this approach to designing services, you would be forgiven for thinking that there is no future for SOAP, or at least architectures that operate along the lines of discrete service operations. The greatest asset of a REST-based architecture is its pure elegance; like all the best ideas, it is devastatingly and beautifully simple. Yet you must consider what it cannot do, which is both its strength and its weakness.

When Microsoft and other big vendors backed SOAP and WS-* standards, all parties were looking for standards that could work in any number of complex scenarios. It is the classic "But what if?" mindset that many architects adopt when they try to build future-proofed systems. The reality of building an extensible model that can deal with all scenarios is that you build complexity into your architecture. This one-size-fits-all bloat is something you definitely feel when you implement services using SOAP, and fans of REST architectures will cry "You ain't gonna need it" from the rooftops.

But be clear here. By choosing to design a service using REST, you are making some big decisions about what you will not need in the future. Table 1-1 looks at key differences between REST services and SOAP/WS-* service capabilities.

Table 1-1. *Capabilities of REST and WS-* Services*

Capability	REST Services	WS-* Services
Addressing	Limited to simple topologies addressed via URI and HTTP verb	Available via WS-Addressing
Transport	HTTP and HTTPS	Any (for example, RPC and TCP/IP)
Encoding	Text only	Any (for example, binary)
Message Exchange Pattern	Request-reply only	Any (for example, duplex)
Topology	Peer to peer	Any (for example, service broker)
Ordering	No support	Available via WS-ReliableMessaging
Transactions	No support	Available via WS-AtomicTransactions
Security	Only transport security via HTTPS	HTTPS, WS-Security, WS-SecureConversation, and WS-Trust

When you look at this table, it is not surprising to see why REST services have become so commonplace for public-facing services on the Internet. In these scenarios, you only ever want to expose your service via HTTP, and you do not want to secure access to your services. You are not interested in atomic transactional support, or the order in which messages are sent and received. You are only interested in supporting simple request-response scenarios. However, services designed for internal consumption within an enterprise or business-to-business can look very different, and some of the additional capabilities of WS-* may be necessary in specific scenarios.

The reality is WS-* is not dead, nor is REST only suitable for public Internet-facing services. As an architect, you have to look at the scenarios you face, determine the probability of change, and choose the appropriate tool for the job. The reality of future enterprise SOAs is that they will mix and match all these technologies, and in the Microsoft space they will share WCF as the common foundation for all services, regardless of which implementation best suits your current needs.

LINQ to Something

So far this chapter has largely focused on describing ADO.NET Data Services from the outside: what ADO.NET Data Services looks like to design and consumes from a messaging perspective. Internal to any service is an implementation, which in ADO.NET Data Services is achieved through a class named `DataService<T>`.

The service contract `IRequestHandler` that `DataService<T>` implements only one operation called `ProcessRequestForMessage`, and the implementation for this method is already written within the `DataService<T>` class as part of ADO.NET Data Services. This operation wires up an HTTP verb and the inbound URI to executing a method in an interface called `IQueryable<T>`, which is defined in the `System.Linq` subsystem of the .NET Framework.

The importance of executing queries against `IQueryable<T>` is that it enables developers to choose which underlying LINQ-based provider they wish to expose as a service through ADO.NET Data Services. Since .NET 3.5 SP1, the framework provides two implementations of `IQueryable<T>`: LINQ to SQL and LINQ to Entities. LINQ to SQL enables you to query a SQL Server database, and LINQ to Entities uses the ADO.NET Entity Framework to enable you to query the conceptual model to manipulate data in an underlying data source. Alternatively, you can also write your own LINQ provider or use some of the many LINQ provider implementations freely available on the Internet, such as LINQ to LDAP for querying an LDAP data source.

Before describing `IQueryable<T>` further, it is important to understand some key technology concepts specific to LINQ.

■ **Note** The general term "LINQ," as mentioned previously, groups together several underlying implementations such as LINQ to Objects, LINQ to SQL, LINQ to Entities, LINQ to Datasets, and LINQ to XML. When you read the term LINQ, it is important to understand which implementation of LINQ is being referred to, as different flavors of LINQ have subtly different characteristics.

Defining LINQ

The beginning of this chapter described the key concept of set-based logic, where a set of data is manipulated using queries (such as those written in SQL). LINQ is a new feature of .NET 3.5 that brings the concepts of set-based logic into the .NET world.

At the heart of LINQ is a framework defined under the `System.Linq` namespace, which when coupled with the new language constructs of C# 3.0 provides the ability to write LINQ queries, either over objects held locally or over remote data sources.

The key features of C# 3.0 that are important to writing LINQ queries are extension methods and lambda expressions. **Extension methods** enable you to extend the methods of an existing type. For example, you could create an extension method to the `String` class called `ToStringArray()`, which when called turns a string variable into a string array. **Lambda expressions** enable you to write an expression within a method signature that is evaluated at runtime. Lambda expressions are a shorthand way of writing a delegate.

LINQ provides classes that contain extension methods commonly referred to as **query operators**. Examples of LINQ query operators are `Select`, `Where`, and `GroupBy`. Many of these methods will feel familiar to SQL developers as much of the set-based logic in LINQ follows the same semantics as SQL. Query operators accept lambda expressions as a way of evaluating the input to the query operator to return an output.

LINQ to Local Data

LINQ defines a class named `Enumerable`, which contains the query operators used to write LINQ queries over local objects in memory. These query operators are extension methods to the `IEnumerable<T>` interface, which is an existing .NET interface inherited by all generic collections, such as `List<T>`. Using extension methods in this way means that any type that inherits `IEnumerable<T>` can be queried using LINQ. Objects that implement `IEnumerable<T>` are collections of objects or XML documents loaded into memory. These two flavors of LINQ are commonly referred to as LINQ to Objects and LINQ to XML, respectively.

Query operators defined in the `Enumerable` class typically accept an input collection and return an output collection. The contents of the output collection would depend on the conditions of the Lambda expression. A simple LINQ query is shown in Listing 1-4.

Listing 1-4. *Example LINQ Query Using Extension Methods and Lambda Expressions*

```
using System;
using System.Linq;

namespace Apress.Data.Web.Samples.NameGenerator
{
    class Program
    {
        static void Main(string[] args)
        {
            string[] names = new string[] { "Simon", "John", "Angharad", "Tim" };

            string[] longnames = GetLongNames(names);

            foreach (string name in longnames)
            {
                Console.WriteLine(name);
            }
        }

        static string[] GetLongNames(string[] names)
        {
            var longnames = names.Where(n => n.Length > 4);

            return longnames.ToArray();
        }
    }
}
```

Executing this code would return two names: Simon and Angharad. The `Where()` method is a query operator defined in `Enumerable`. The lambda expression it executes returns an output collection of string objects longer than four characters. The `n` variable is of type `string` because the array `names` is a string array. The `Where()` query operator is able to be executed directly from the `names` string array, because the string array inherits `IEnumerable<T>`.

Notice that the object `longnames` is declared as type `var`. The `var` keyword in C# 3.0 is an implicit type. This means that it is strongly typed, taking the return type of the `Where()` method, which is the output collection of type `IEnumerable<string>`. Finally, the query operator `ToArray()` is called to cast the output sequence to the type `string[]`. This is an important step in the preceding code, because it is only here that the query is actually executed; this process is known as **deferred execution**, which can mislead developers who think that the result of the query is set on the line of code that returns the original output sequence.

LINQ to Remote Data

LINQ defines a class named `Queryable`, which contains the query operators used to write LINQ queries to remote data. These query operators are extension methods to the `IQueryable<T>` interface, which is an interface inherited by collections that **represent** the remote data source. The query operators under the `Queryable` class largely mirror those in the `Enumerable` class, meaning that to developers, the syntax for writing LINQ queries is the same whether you are querying local objects or remote data.

The `IQueryable<T>` objects that represent the underlying data source are objects provided by a LINQ implementation. The two LINQ implementations that come out the box with the .NET Framework are LINQ to SQL and LINQ to Entities. LINQ to SQL translates LINQ queries into Transact SQL (T-SQL) for SQL Server. LINQ to Entities translate a LINQ query into Entity SQL (E-SQL) for the ADO.NET Entity Framework. E-SQL is a special form of SQL used by the ADO.NET Entity Framework to query the conceptual model defined by the ORM.

■**Note** It is possible to write your own LINQ provider by inheriting `IQueryable<T>`. Alternatively, as mentioned previously, many LINQ providers are freely available on the Internet, such as LINQ to LDAP, used to translate LINQ queries into LDAP queries.

A LINQ implementation provides `IQueryable<T>` classes containing implementation details to interpret a query for a specific data source. For example, LINQ to SQL provides classes such as `Table<T>` to represent tables of data in SQL Server, providing specific methods to execute against a table of SQL Server data.

Using LINQ to query remote data sources is very similar to querying in-memory objects, but for one extra task; you must develop special entity classes that represent the entities stored in your underlying database. These entities map C# types to the underlying data structure. In LINQ to SQL, this is achieved by attaching attributes to the types in your entity, specifying the columns and key constraints in the underlying tables. This metadata is used by LINQ to SQL to understand how to translate the LINQ query into T-SQL.

The extra work of generating entity types for your data source is alleviated by the provision of tools to automatically generate these classes based on the structure of the data source itself. From within Visual Studio you can generate LINQ to SQL classes from a SQL Server database and use the visual designer to manipulate the created classes. Similarly, the ADO.NET Entity Framework provides a designer for creating a conceptual model of your entities, which inherit from `IQueryable<T>`.

The `IQueryable<T>` classes generated either by using LINQ to SQL or LINQ to Entities are wrapped in a context class (named `DataContext` for LINQ to SQL or `ObjectContext` for LINQ to

Entities). It is this class that is wired into the ADO.NET Data Services `DataService<T>` service implementation, where `T` is your strongly typed context object. The WCF service implementation of ADO.NET Data Services translates the inbound URI and HTTP verb to a LINQ query against your chosen `IQueryable` data source within the `ProcessRequestForMessage` method. This LINQ query is then interpreted by the underlying LINQ implementation to execute the query native to the data source.

LINQ TO SQL VS. LINQ TO ENTITIES

You may be wondering at this point why there are two implementations of LINQ that work for databases in the .NET Framework? Either will work with ADO.NET Data Services, but which one should you choose?

The key difference between LINQ to SQL and LINQ to Entities is that LINQ to Entities provides a conceptual model for your database, whereas LINQ to SQL works much more closely with the relational data model of your database. The extra layer of abstraction LINQ to Entities provides from the structure of the database itself enables you to design an object-oriented representation (conceptual model) of your relational database. Queries written in LINQ to Entities are executed against this conceptual model using E-SQL. The entity framework handles mapping between the conceptual model of your entities and the relational model in your database using an ORM, thus addressing an impedance mismatch that can occur between how you ideally want to model the data in the database and how you want represent an object model in your code.

LINQ to SQL has limited mapping capabilities and unlike LINQ to Entities only works with SQL Server. However, if you do not have a big impedance mismatch between your database design and your object model and you are using SQL Server, LINQ to SQL is a simpler and therefore quicker option to use for development.

Summary

This chapter covers a lot of ground in the .NET field because ADO.NET Data Services touches so many concepts and technologies. Clearly, there is much to take in and think about here: the ramifications of REST on the world of service orientation, and the contribution of both WCF and LINQ to the architecture of ADO.NET Data Services and the wider enterprise.

ADO.NET Data Services is a great example of the whole being more valuable than the sum of its parts; the concepts of REST and the implementation of ADO.NET Data Services are devastatingly elegant. When you write your first REST service and come to terms with what it really means to your world, you may well ask yourself, “What do I do now?” It does indeed feel so very simple, but it is always possible to create a bad design regardless of the technologies you are using.

Developing services using ADO.NET Data Services enables you to focus on important aspects of service design that in the past may have been ignored due to the amount of heavy lifting needed elsewhere. Focus on the relational database design to ensure the best structure for your data store. Focus on the object model design to ensure the most natural programming model. Focus on the design of your service for easy client consumption by targeting the best data formats and URI structures. Finally, learn when and when not to apply a REST service architecture.