Developing Windows Azure and Web Services

Exam Ref 70-487

William Ryan
Wouter de Kort
Shane Milton

Exam Ref 70-487

Prepare for Microsoft Exam 70-487—and help demonstrate your real-world mastery of developing Windows Azure and web services. Designed for experienced developers ready to advance their status, *Exam Ref* focuses on the critical-thinking and decision-making acumen needed for success at the Microsoft Specialist level.

Focus on the expertise measured by these objectives:

- Accessing data
- Querying and manipulating data by using the Entity Framework
- Designing and implementing WCF Services
- Creating and consuming Web API-based services
- Deploying web applications and services

This Microsoft *Exam Ref:*

- Organizes its coverage by exam objectives.
- Features strategic, what-if scenarios to challenge you.
- Includes a 15% exam discount from Microsoft. Offer expires 12/31/2015. Details inside.

Developing Windows Azure and Web Services

About the Exam

Exam 70-487 is one of three Microsoft exams focused on the skills and knowledge necessary to create and deploy modern web applications and services.

About Microsoft Certification

Passing this exam earns you a **Microsoft Specialist** certification. You also receive credit toward a **Microsoft Certified Solutions Developer (MCSD)** certification that proves your ability to build innovative solutions across multiple technologies, both on-premises and in the cloud.

Exams 70-480, 70-486, and 70-487 are required for the MCSD: Web Applications certification.

See full details at:
microsoft.com/learning/certification

About the Authors

William Ryan is a solution architect who has served as a subject matter expert for several Microsoft exams, including Microsoft SQL Server and Windows Azure.

Wouter de Kort, MCSD, is an independent technical coach, trainer, and developer who works with C# and .NET, Entity Framework, and Team Foundation Server.

Shane Milton is a senior architect creating enterprise systems running in Windows Azure. He also designs cloud-based solutions for managing energy in homes and businesses.

microsoft.com/mspress

ISBN: 978-0-7356-7724-1

U.S.A. $39.99
Canada $41.99
[*Recommended*]

Certification/Windows Server

Microsoft

Exam Ref 70-487: Developing Windows Azure and Web Services

William Ryan
Wouter de Kort
Shane Milton

Published with the authorization of Microsoft Corporation by:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, California 95472

Copyright © 2013 by O'Reilly Media, Inc.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

ISBN: 978-0-7356-7724-1

1 2 3 4 5 6 7 8 9 QG 8 7 6 5 4 3

Printed and bound in the United States of America.

Microsoft Press books are available through booksellers and distributors worldwide. If you need support related to this book, email Microsoft Press Book Support at *mspinput@microsoft.com*. Please tell us what you think of this book at *http://www.microsoft.com/learning/booksurvey*.

Microsoft and the trademarks listed at *http://www.microsoft.com/about/legal/en/us/IntellectualProperty/Trademarks/EN-US.aspx* are trademarks of the Microsoft group of companies. All other marks are property of their respective owners.

The example companies, organizations, products, domain names, email addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, email address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, O'Reilly Media, Inc., Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Jeff Riley

Developmental Editor: Ginny Bess Munroe

Production Editor: Kara Ebrahim

Editorial Production: Box Twelve Communications

Technical Reviewer: Shane Milton

Copyeditor: Nancy Sixsmith

Indexer: Angie Martin

Cover Design: Twist Creative • Seattle

Cover Composition: Ellie Volckhausen

Illustrator: Rebecca Demarest

Contents at a glance

	Introduction	*xv*
CHAPTER 1	**Accessing data**	**1**
CHAPTER 2	**Querying and manipulating data by using the Entity Framework**	**111**
CHAPTER 3	**Designing and implementing WCF Services**	**169**
CHAPTER 4	**Creating and consuming Web API-based services**	**287**
CHAPTER 5	**Deploying web applications and services**	**361**
	Index	*437*

Contents

Introduction **xv**

Microsoft certifications *xv*

Acknowledgments *xvi*

Errata & book support *xvi*

We want to hear from you *xvi*

Stay in touch *xvi*

Preparing for the exam *xvii*

Chapter 1 Accessing data **1**

Objective 1.1: Choose data access technologies 1

Choosing a technology (ADO.NET, Entity Framework, WCF Data Services) based on application requirements 1

Choosing EF as the data access technology 11

Choosing WCF Data Services as the data access technology 31

Objective summary 35

Objective review 35

Objective 1.2: Implement caching 36

Understanding caching options 37

Using the ObjectCache 38

Using the HttpContext.Cache 43

Objective summary 51

Objective review 52

Objective 1.3: Implement transactions 53

Understanding characteristics of transactions 53

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Implementing distributed transactions 54
Specifying a transaction isolation level 55
Managing transactions by using the API from the System.Transactions namespace 57
Using the EntityTransaction 58
Using the SqlTransaction 59
Objective summary 60
Objective review 60

Objective 1.4: Implement data storage in Windows Azure 61
Accessing data storage in Windows Azure 61
Choosing a data storage mechanism in Windows Azure (blobs, tables, queues and SQL Database) 64
Distribute data by using the Windows Azure Content Delivery Network (CDN) 69
Manage Windows Azure Caching 71
Handling exceptions by using retries (SQL Database) 72
Objective summary 74
Objective review 75

Objective 1.5: Create and implement a WCF Data Services service. 75
Addressing resources 76
Creating a query 79
Accessing payload formats 83
Working with interceptors and service operators 83
Objective summary 85
Objective review 86

Objective 1.6: Manipulate XML data structures. 86
Reading, filtering, creating, and modifying XML structures 87
Manipulating XML data 90
XPath 95
LINQ-to-XML 96
Advanced XML manipulation 100
Objective summary 102
Objective review 103

Chapter summary 103

Answers. 105

Chapter 2 Querying and manipulating data by using the Entity Framework 111

Objective 2.1: Query and manipulate data by using the Entity Framework 111

Querying, updating, and deleting data by using DbContext 112
Building a query that uses deferred execution 113
Implementing lazy loading and eager loading 115
Creating and running compiled queries 118
Querying data by using Entity SQL 119
Objective summary 121
Objective review 121

Objective 2.2: Query and manipulate data by using Data Provider for Entity Framework 122

Querying and manipulating data by using Connection, DataReader, Command from the System.Data.EntityClient namespace 122
Performing synchronous and asynchronous operations 124
Managing transactions (API) 124
Objective summary 126
Objective review 126

Objective 2.3: Query data by using LINQ to Entities 127

Querying data using LINQ operators 128
IEnumerable versus IQueryable 129
Logging queries 129
Objective summary 130
Objective review 131

Objective 2.4: Query and manipulate data by using ADO.NET 131

Querying data using Connection, DataReader, Command, DataAdapter, and DataSet 132
SqlConnection 132
SqlCommand 133
SqlDataReader 134
Performing synchronous and asynchronous operations 141
Managing transactions 142

Objective summary 143
Objective review 144

Objective 2.5: Create an Entity Framework data model. 145
Structuring the data model using Table-per-Type and Table-per-Hierarchy inheritance 145
Choosing and implementing an approach to manage a data model (code first vs. model first vs. database first) 146
Implementing POCOs 153
Describing a data model using conceptual schema definitions, storage schema definitions, and mapping language (CSDL, SSDL, & MSL) 156
Objective summary 160
Objective review 160

Chapter summary 161

Answers. 162

Chapter 3 Designing and implementing WCF Services 169

Objective 3.1: Create a WCF service 170
Defining SOA concepts 170
Creating contracts 171
Implementing inspectors 192
Implementing message inspectors 194
Objective summary 197
Objective review 198

Objective 3.2: Configure WCF services by using configuration settings 199
Configuring service behaviors 200
Creating a new service 200
Specifying a new service element (service) 201
Specifying a new service element (contract) 202
Specifying a new service element (communication mode) 203
Specifying a new service element (interoperability mode) 203
Resulting configuration file 204
Exposing service metadata 205

Objective summary 211
Objective review 212

Objective 3.3: Configure WCF services by using the API 212
Configuring service endpoints 213
Configuring service behaviors 214
Configuring bindings 217
Specifying a service contract 221
Objective summary 225
Objective review 226

Objective 3.4: Secure a WCF service 227
Implementing message level security 227
Implementing transport level security 229
Implementing certificates 230
Objective summary 231
Objective review 232

Objective 3.5: Consume WCF services. 233
Generating proxies using Svcutil.exe 233
Generating proxies by creating a service reference 235
Creating and implementing channel factories 239
Objective summary 242
Objective review 243

Objective 3.6: Version a WCF service. 244
Versioning different types of contracts 244
Configuring address, binding, and routing service versioning 246
Objective summary 247
Objective review 247

Objective 3.7: Create and configure a WCF service on Windows Azure 249
Creating and configuring bindings for WCF services 249
Relaying bindings to Azure using service bus endpoints 252
Integrating with the Azure service bus relay 252
Objective summary 254
Objective review 254

Objective 3.8: Implement messaging patterns 255
Implementing one-way, request/reply, streaming, and duplex communication 256
Implementing Windows Azure service bus and Windows Azure queues 260
Objective summary 262
Objective review 263
Objective 3.9: Host and manage services. 264
Managing services concurrency 264
Choosing an instancing mode 265
Creating service hosts 266
Choosing a hosting mechanism 270
Creating transactional services 271
Hosting services in a Windows Azure worker role 272
Objective summary 273
Objective review 274
Chapter summary 275
Answers. 276
Chapter 4 Creating and consuming Web API-based services 287
Objective 4.1: Design a Web API 287
Choosing appropriate HTTP methods 288
Mapping URI space using routing 299
Choosing appropriate formats for responses to meet requirements 304
Planning when to make HTTP actions asynchronous 304
Objective summary 306
Objective review 307
Objective 4.2: Implement a Web API. 308
Accepting data in JSON format 308
Using content negotiation to deliver different data formats 312
Defining actions and parameters to handle data binding 315
Using HttpMessageHandler to process client requests and server responses 316
Implementing dependency injection 317

Implementing action filters and exception filters 320
Implementing asynchronous and synchronous actions 321
Implementing streaming actions 321
Objective summary 323
Objective review 324

Objective 4.3: Secure a Web API 324
Authenticating and authorizing users 325
Implementing HttpBasic authentication 326
Implementing Windows Authentication 329
Preventing cross-site request forgery 330
Enabling cross-domain requests 333
Implementing and extending authorization filters 334
Objective summary 336
Objective review 336

Objective 4.4: Host and manage a Web API 337
Self-hosting a Web API 338
Hosting Web API in an ASP.NET app 340
Hosting services in a Windows Azure worker role 341
Restricting message size 342
Configuring the host server for streaming 343
Objective summary 345
Objective review 345

Objective 4.5: Consume Web API web services. 346
Consuming Web API services 346
Sending and receiving requests in different formats 350
Objective summary 352
Objective review 352

Chapter summary 353

Answers. 354

Chapter 5 Deploying web applications and services 361

Objective 5.1: Design a deployment strategy 362
Deploying a web application by using XCopy 362
Creating an IIS install package 367

Automating a deployment from TFS or Build Server 367
Deploying to web farms 371
Objective summary 373
Objective review 373

Objective 5.2: Choose a deployment strategy for a Windows Azure web application 374
Performing an in-place upgrade and VIP Swap 374
Configuring an upgrade domain 375
Upgrading through a VIP Swap 376
Creating and configuring input and internal endpoints 377
Specifying operating system configuration 380
Objective summary 382
Objective review 383

Objective 5.3: Configure a web application for deployment 383
Switching from production/release mode to debug 384
Transforming web.config by XSLT 385
Using SetParameters to set up an IIS app pool 387
Configuring Windows Azure configuration settings 390
Objective summary 392
Objective review 392

Objective 5.4: Manage packages by using NuGet 393
Installing and updating an existing NuGet package 394
Creating and configuring a NuGet package 399
Setting up your own package repository 403
Objective summary 405
Objective review 406

Objective 5.5: Create, configure, and publish a web package 406
Creating an IIS InstallPackage 407

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Configuring the build process to output a web package 415
Applying pre- and post-condition actions 416
Objective summary 417
Objective review 417

Objective 5.6: Share assemblies between multiple applications and servers 418
Preparing the environment for use of assemblies across multiple servers 419
Signing assemblies by using a strong name 420
Deploying assemblies to the global assembly cache 422
Implementing assembly versioning 423
Creating an assembly manifest 426
Objective summary 427
Objective review 428

Chapter summary 429

Answers. 430

Index *437*

Introduction

Most books take a low-level approach, teaching you how to use individual classes and how to accomplish granular tasks. Like other Microsoft certification exams, this book takes a high-level approach, building on your knowledge of lower-level Microsoft Windows application development and extending it into application design. Both the exam and the book are so high level that there is little coding involved. In fact, most of the code samples in this book illustrate higher-level concepts.

The exam is written for developers who have three to five years of experience developing Web Services and at least one year of experience developing Web API and Azure solutions. Developers should also have at least three years of experience working with Relational Database Management systems and ADO.NET and at least one year of experience with the Entity Framework.

This book covers every exam objective, but it does not cover every exam question. Only the Microsoft exam team has access to the exam questions themselves, and Microsoft regularly adds new questions to the exam, making it impossible to cover specific questions. You should consider this book a supplement to your relevant real-world experience and other study materials. If you encounter a topic in this book that you do not feel completely comfortable with, use the links you'll find in the text to find more information and take the time to research and study the topic. Valuable information is available on MSDN, TechNet, and in blogs and forums.

Microsoft certifications

Microsoft certifications distinguish you by proving your command of a broad set of skills and experience with current Microsoft products and technologies. The exams and corresponding certifications are developed to validate your mastery of critical competencies as you design and develop, or implement and support, solutions with Microsoft products and technologies both on-premise and in the cloud. Certification brings a variety of benefits to the individual and to employers and organizations.

MORE INFO **ALL MICROSOFT CERTIFICATIONS**

For information about Microsoft certifications, including a full list of available certifications, go to *http://www.microsoft.com/learning/en/us/certification/cert-default.aspx*.

Acknowledgments

I'd like to thank Ginny Munroe and Shane Milton for the immense help they provided in preparing this book. My wife and daughter were extremely supportive throughout this stressful and difficult time. I'd also like to thank Walter Bellhaven and Herb Sewell for always keeping things uplifting.

Errata & book support

We've made every effort to ensure the accuracy of this book and its companion content. Any errors that have been reported since this book was published are listed on our Microsoft Press site at oreilly.com:

http://aka.ms/ER70-487/errata

If you find an error that is not already listed, you can report it to us through the same page.

If you need additional support, email Microsoft Press Book Support at *mspinput@microsoft.com*.

Please note that product support for Microsoft software is not offered through the addresses above.

We want to hear from you

At Microsoft Press, your satisfaction is our top priority, and your feedback our most valuable asset. Please tell us what you think of this book at:

http://www.microsoft.com/learning/booksurvey

The survey is short, and we read every one of your comments and ideas. Thanks in advance for your input!

Stay in touch

Let's keep the conversation going! We're on Twitter: *http://twitter.com/MicrosoftPress*.

Preparing for the exam

Microsoft certification exams are a great way to build your resume and let the world know about your level of expertise. Certification exams validate your on-the-job experience and product knowledge. While there is no substitution for on-the-job experience, preparation through study and hands-on practice can help you prepare for the exam. We recommend that you round out your exam preparation plan by using a combination of available study materials and courses. For example, you might use the Exam Ref and another study guide for your "at home" preparation, and take a Microsoft Official Curriculum course for the classroom experience. Choose the combination that you think works best for you.

Note that this Exam Ref is based on publically available information about the exam and the author's experience. To safeguard the integrity of the exam, authors do not have access to the live exam.

CHAPTER 1

Accessing data

It's hard to find a modern software application that doesn't make extensive use of data access. Some exist, but particularly in the business realm, most have a heavy data access component. There are many ways to build data-centric applications and many technologies that can be used. Microsoft provides several, including ADO.NET, Entity Framework, and SQL Server. This objective covers about 24 percent of the exam's questions.

> ***important***
> ***Have you read page xvii?***
> **It contains valuable information regarding the skills you need to pass the exam.**

Objectives in this chapter:

- Objective 1.1: Choose data access technologies
- Objective 1.2: Implement caching
- Objective 1.3: Implement transactions
- Objective 1.4: Implement data storage in Windows Azure
- Objective 1.5: Create and implement a WCF Data Services service
- Objective 1.6: Manipulate XML data structures

Objective 1.1: Choose data access technologies

There's no law that states that only one data access technology must be used per application. However, unless you have a specific need, it's generally advisable to pick a data access technology and stick with it throughout the application. Three obvious choices covered by this exam are *ADO.NET*, *Entity Framework (EF)*, and *WCF Data Services*.

This objective covers how to:

- Choose a technology (ADO.NET, Entity Framework, WCF Data Services) based on application requirements

Choosing a technology (ADO.NET, Entity Framework, WCF Data Services) based on application requirements

Choosing a data access technology is something that requires thought. For the majority of cases, anything you can do with one technology can be accomplished with the other technologies. However, the upfront effort can vary considerably. The downstream benefits and costs are generally more profound. WCF Data Services might be overkill for a simple one-user scenario. A console application that uses ADO.NET might prove much too limiting for any multiuser scenario. In any case, the decision of which technology to use should not be undertaken lightly.

Choosing ADO.NET as the data access technology

If tasked to do so, you could write a lengthy paper on the benefits of using ADO.NET as a primary data access technology. You could write an equally long paper on the downsides of using ADO.NET. Although it's the oldest of the technologies on the current stack, it still warrants serious consideration, and there's a lot to discuss because there's a tremendous amount of ADO.NET code in production, and people are still using it to build new applications.

ADO.NET was designed from the ground up with the understanding that it needs to be able to support large loads and to excel at security, scalability, flexibility, and dependability. These performance-oriented areas (security, scalability, and so on) are mostly taken care of by the fact that ADO.NET has a bias toward a *disconnected model* (as opposed to ADO's commonly used *connected model*). For example, when using individual commands such as INSERT, UPDATE, or DELETE statements, you simply open a connection to the database, execute the command, and then close the connection as quickly as possible. On the query side, you create a SELECT query, pull down the data that you need to work with, and immediately close the connection to the database after the query execution. From there, you'd work with a localized version of the database or subsection of data you were concerned about, make any changes to it that were needed, and then submit those changes back to the database (again by opening a connection, executing the command, and immediately closing the connection).

There are two primary reasons why a connected model versus disconnected model is important. First of all, connections are expensive for a relational database management system (RDBMS) to maintain. They consume processing and networking resources, and database systems can maintain only a finite number of active connections at once. Second, connections can hold locks on data, which can cause concurrency problems. Although it doesn't solve all your problems, keeping connections closed as much as possible and opening them only for short periods of time (the absolute least amount of time possible) will go a long way to mitigating many of your database-focused performance problems (at least the problems caused by the consuming application; database administrator (DBA) performance problems are an entirely different matter).

To improve efficiency, ADO.NET took it one step farther and added the concept of *connection pooling*. Because ADO.NET opens and closes connections at such a high rate, the minor overheads in establishing a connection and cleaning up a connection begin to affect

performance. Connection pooling offers a solution to help combat this problem. Consider the scenario in which you have a web service that 10,000 people want to pull data from over the course of 1 minute. You might consider immediately creating 10,000 connections to the database server the moment the data was requested and pulling everybody's data all at the same time. This will likely cause the server to have a meltdown! The opposite end of the spectrum is to create one connection to the database and to make all 10,000 requests use that same connection, one at a time.

Connection pooling takes an in-between approach that works much better. It creates a few connections (let's say 50). It opens them up, negotiates with the RDBMS about how it will communicate with it, and then enables the requests to share these active connections, 50 at a time. So instead of taking up valuable resources performing the same nontrivial task 10,000 times, it does it only 50 times and then efficiently funnels all 10,000 requests through these 50 channels. This means each of these 50 connections would have to handle 200 requests in order to process all 10,000 requests within that minute. Following this math, this means that, if the requests can be processed on average in under ~300ms, you can meet this requirement. It can take ~100ms to open a new connection to a database. If you included that within that 300ms window, 33 percent of the work you have to perform in this time window is dedicated simply to opening and closing connections, and that will never do!

Finally, one more thing that connection pooling does is manage the number of active connections for you. You can specify the maximum number of connections in a connection string. With an ADO.NET 4.5 application accessing SQL Server 2012, this limit defaults to 100 simultaneous connections and can scale anywhere between that and 0 without you as a developer having to think about it.

ADO.NET compatibility

Another strength of ADO.NET is its cross-platform compatibility. It is compatible with much more than just SQL Server. At the heart of ADO.NET is the System.Data namespace. It contains many base classes that are used, irrespective of the RDBMS system. There are several vendor-specific libraries available (System.Data.SqlClient or System.Data.OracleClient, for instance) as well as more generic ones (System.Data.OleDb or System.Data.Odbc) that enable access to OleDb and Odbc-compliant systems without providing much vendor-specific feature access.

ADO.NET architecture

The following sections provide a quick overview of the ADO.NET architecture and then discuss the strengths and benefits of using it as a technology. A few things have always been and probably always will be true regarding database interaction. In order to do anything, you need to connect to the database. Once connected, you need to execute commands against the database. If you're manipulating the data in any way, you need something to hold the data that you just retrieved from the database. Other than those three constants, everything else can have substantial variability.

> ***NOTE*** **PARAMETERIZE YOUR QUERIES**
>
> **There is no excuse for your company or any application you work on to be hacked by an injection attack (unless hackers somehow find a vulnerability in the DbParameter class that's been heretofore unknown). Serious damage to companies, individual careers, and unknowing customers has happened because some developer couldn't be bothered to clean up his dynamic SQL statement and replace it with parameterized code. Validate all input at every level you can, and at the same time, make sure to parameterize everything as much as possible. This one of the few serious bugs that is always 100 percent avoidable, and if you suffer from it, it's an entirely self-inflicted wound.**

.NET Framework data providers

According to MSDN, .NET Framework *data providers* are described as "components that have been explicitly designed for data manipulation and fast, forward-only, read-only access to data." Table 1-1 lists the foundational objects of the data providers, the base class they derive from, some example implementations, and discussions about any relevant nuances.

TABLE 1-1 .NET Framework data provider overview

Provider object	Interface	Example items	Discussion
DbConnection	IDbConnection	SqlConnection, OracleConnection, EntityConnection, OdbcConnection, OleDbConnection	Necessary for any database interaction. Care should be taken to close connections as soon as possible after using them.
DbCommand	IDbCommand	SqlCommand, OracleCommand, EntityCommand, OdbcCommand, OleDbCommand	Necessary for all database interactions in addition to Connection. Parameterization should be done only through the Parameters collection. Concatenated strings should never be used for the body of the query or as alternatives to parameters.
DbDataReader	IDataReader	SqlDataReader, OracleDataReader, EntityDataReader, OdbcDataReader, OleDbDataReader	Ideally suited to scenarios in which speed is the most critical aspect because of its forward-only nature, similar to a Stream. This provides read-only access to the data.
DbDataAdapter	IDbDataAdapter	SqlDataAdapter, OracleDataAdapter, OdbcDataAdapter, OleDbDataAdapter	Used in conjunction with a Connection and Command object to populate a DataSet or an individual DataTable, and can also be used to make modifications back to the database. Changes can be batched so that updates avoid unnecessary roundtrips to the database.

Provider object	Interface	Example items	Discussion
DataSet	N/A	No provider-specific implementation	In-memory copy of the RDBMS or portion of RDBMS relevant to the application. This is a collection of DataTable objects, their relationships to one another, and other metadata about the database and commands to interact with it.
DataTable	N/A	No provider-specific implementation	Corresponds to a specific view of data, hether from a SELECT query or generated from .NET code. This is often analogous to a table in the RDBMS, although only partially populated. It tracks the state of data stored in it so, when data is modified, you can tell which records need to be saved back into the database.

The list in Table 1-1 is not a comprehensive list of the all the items in the System.Data (and provider-specific) namespace, but these items do represent the core foundation of ADO.NET. A visual representation is provided in Figure 1-1.

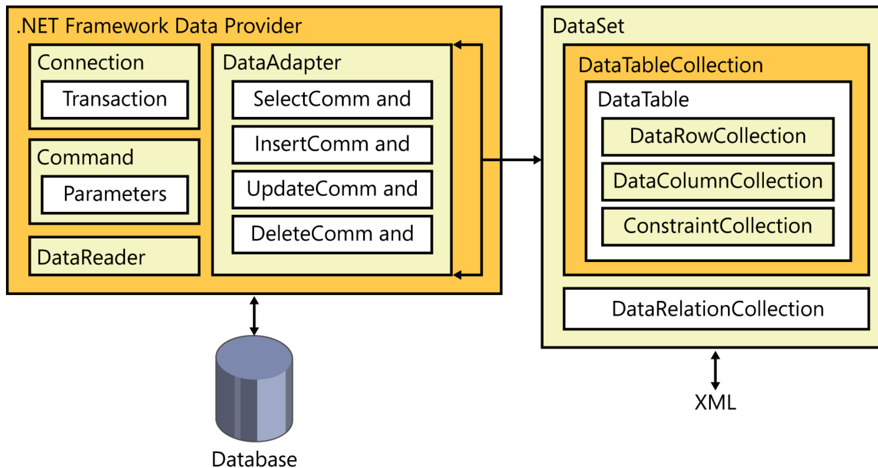

FIGURE 1-1 .NET Framework data provider relationships

DataSet or DataReader?

When querying data, there are two mechanisms you can use: a *DataReader* or a *DataAdapter*. These two options are more alike than you might think. This discussion focuses on the differences between using a DataReader and a DataAdapter, but if you said, “Every SELECT query operation you employ in ADO.NET uses a DataReader,” you’d be correct. In fact, when you use a DataAdapter and something goes wrong that results in an exception being thrown, you’ll typically see something like the following in the StackTrace of the exception: “System. InvalidOperationException: ExecuteReader requires an open and available Connection.” This

exception is thrown after calling the Fill method of a SqlDataAdapter. Underneath the abstractions, a DataAdapter uses a DataReader to populate the returned DataSet or DataTable.

Using a DataReader produces faster results than using a DataAdapter to return the same data. Because the DataAdapter actually uses a DataReader to retrieve data, this should not surprise you. But there are many other reasons as well. Look, for example, at a typical piece of code that calls both:

```
[TestCase(3)]
public static void GetCustomersWithDataAdapter(int customerId)
{
    // ARRANGE
    DataSet customerData = new DataSet("CustomerData");
    DataTable customerTable = new DataTable("Customer");
    customerData.Tables.Add(customerTable);

    StringBuilder sql = new StringBuilder();
    sql.Append("SELECT FirstName, LastName, CustomerId, AccountId");
    sql.Append("  FROM [dbo].[Customer] WHERE CustomerId = @CustomerId ");

    // ACT
    // Assumes an app.config file has connectionString added to <connectionStrings>
section named "TestDB"
    using (SqlConnection mainConnection =
        new SqlConnection(ConfigurationManager.ConnectionStrings["TestDB"].
ConnectionString))
    {
       using (SqlCommand customerQuery = new SqlCommand(sql.ToString(), mainConnection))
        {
            customerQuery.Parameters.AddWithValue("@CustomerId", customerId);
            using (SqlDataAdapter customerAdapter = new SqlDataAdapter(customerQuery))
            {
                try
                {
                    customerAdapter.Fill(customerData, "Customer");
                }
                finally
                {
                    // This should already be closed even if we encounter an exception
                    // but making it explicit in code.
                    if (mainConnection.State != ConnectionState.Closed)
                    {
                        mainConnection.Close();
                    }
                }
            }
        }
    }

    // ASSERT
    Assert.That(customerTable.Rows.Count, Is.EqualTo(1), "We expected exactly 1 record
to be returned.");
    Assert.That(customerTable.Rows[0].ItemArray[customerTable.Columns["customerId"].
Ordinal],
        Is.EqualTo(customerId), "The record returned has an ID different than
```

```
expected.");
}
```

Query of Customer Table using SqlDataReader

```
[TestCase(3)]
public static void GetCustomersWithDataReader(int customerId)
{
    // ARRANGE
    // You should probably use a better data structure than a Tuple for managing your
data.
    List<Tuple<string, string, int, int>> results = new List<Tuple<string, string, int,
int>>();

    StringBuilder sql = new StringBuilder();
    sql.Append("SELECT FirstName, LastName, CustomerId, AccountId");
    sql.Append("  FROM [dbo].[Customer] WHERE CustomerId = @CustomerId ");

    // ACT
    // Assumes an app.config file has connectionString added to <connectionStrings>
section named "TestDB"
    using (SqlConnection mainConnection =
        new SqlConnection(ConfigurationManager.ConnectionStrings["TestDB"].
ConnectionString))
    {
        using (SqlCommand customerQuery = new SqlCommand(sql.ToString(),
mainConnection))
        {
            customerQuery.Parameters.AddWithValue("@CustomerId", customerId);
            mainConnection.Open();
            using (SqlDataReader reader = customerQuery.ExecuteReader(CommandBehavior.
CloseConnection))
            {
                try
                {
                    int firstNameIndex = reader.GetOrdinal("FirstName");
                    int lastNameIndex = reader.GetOrdinal("LastName");
                    int customerIdIndex = reader.GetOrdinal("CustomerId");
                    int accountIdIndex = reader.GetOrdinal("AccountId");

                    while (reader.Read())
                    {
                        results.Add(new Tuple<string, string, int, int>(
                          (string)reader[firstNameIndex], (string)reader[lastNameIndex],
                            (int)reader[customerIdIndex], (int)reader[accountIdIndex]));
                    }
                }
                finally
                {
                    // This will soon be closed even if we encounter an exception
                    // but making it explicit in code.
                    if (mainConnection.State != ConnectionState.Closed)
                    {
                        mainConnection.Close();
                    }
                }
            }
```

```
        }
    }

    // ASSERT
    Assert.That(results.Count, Is.EqualTo(1), "We expected exactly 1 record to be
returned.");
    Assert.That(results[0].Item3, Is.EqualTo(customerId),
        "The record returned has an ID different than expected.");
}
```

Test the code and note the minimal differences. They aren't identical functionally, but they are close. The DataAdapter approach takes approximately 3 milliseconds (ms) to run; the DataReader approach takes approximately 2 ms to run. The point here isn't that the DataAdapter approach is 50 percent slower; it is approximately 1 ms slower. Any data access times measured in single-digit milliseconds is about as ideal as you can hope for in most circumstances. Something else you can do is use a profiling tool to monitor SQL Server (such as SQL Server Profiler) and you will notice that both approaches result in an identical query to the database.

> ***IMPORTANT*** **MAKE SURE THAT YOU CLOSE EVERY CONNECTION YOU OPEN**
>
> **To take advantage of the benefits of ADO.NET, unnecessary connections to the database must be minimized. Countless hours, headaches, and much misery result when a developer takes a shortcut and doesn't close the connections. This should be treated as a Golden Rule: If you open it, close it. Any command you use in ADO.NET outside of a DataAdapter requires you to specifically open your connection. You must take explicit measures to make sure that it is closed. This can be done via a try/catch/finally or try/finally structure, in which the call to close the connection is included in the finally statement. You can also use the Using statement (which originally was available only in C#, but is now available in VB.NET), which ensures that the Dispose method is called on IDisposable objects. Even if you use a Using statement, an explicit call to Close is a good habit to get into. Also keep in mind that the call to Close should be put in the finally block, not the catch block, because the Finally block is the only one guaranteed to be executed according to Microsoft.**

The following cases distinguish when you might choose a DataAdapter versus a DataReader:

- Although coding styles and technique can change the equation dramatically, as a general rule, using a DataReader results in faster access times than a DataAdapter does. (This point can't be emphasized enough: The actual code written can and will have a pronounced effect on overall performance.) Benefits in speed from a DataReader can easily be lost by inefficient or ineffective code used in the block.

- DataReaders provide multiple asynchronous methods that can be employed (BeginExecuteNonQuery, BeginExecuteReader, BeginExecuteXmlReader). DataAdapters on the other hand, essentially have only synchronous methods. With small-sized record sets, the differences in performance or advantages of using asynchronous methods are trivial. On large queries that take time, a DataReader, in conjunction with asynchronous methods, can greatly enhance the user experience.
- The Fill method of DataAdapter objects enables you to populate only DataSets and DataTables. If you're planning to use a custom business object, you have to first retrieve the DataSet or DataTables; then you need to write code to hydrate your business object collection. This can have an impact on application responsiveness as well as the memory your application uses.
- Although both types enable you to execute multiple queries and retrieve multiple return sets, only the DataSet lets you closely mimic the behavior of a relational database (for instance, add Relationships between tables using the Relations property or ensure that certain data integrity rules are adhered to via the EnforceConstraints property).
- The Fill method of the DataAdapter completes only when all the data has been retrieved and added to the DataSet or DataTable. This enables you to immediately determine the number of records in any given table. By contrast, a DataReader can indicate whether data was returned (via the HasRows property), but the only way to know the exact record count returned from a DataReader is to iterate through it and count it out specifically.
- You can iterate through a DataReader only once and can iterate through it only in a forward-only fashion. You can iterate through a DataTable any number of times in any manner you see fit.
- DataSets can be loaded directly from XML documents and can be persisted to XML natively. They are consequently inherently serializable, which affords many features not natively available to DataReaders (for instance, you can easily store a DataSet or a DataTable in Session or View State, but you can't do the same with a DataReader). You can also easily pass a DataSet or DataTable in between tiers because it is already serializable, but you can't do the same with a DataReader. However, a DataSet is also an expensive object with a large memory footprint. Despite the ease in doing so, it is generally ill-advised to store it in Session or Viewstate variables, or pass it across multiple application tiers because of the expensive nature of the object. If you serialize a DataSet, proceed with caution!
- After a DataSet or DataTable is populated and returned to the consuming code, no other interaction with the database is necessary unless or until you decide to send the localized changes back to the database. As previously mentioned, you can think of the dataset as an in-memory copy of the relevant portion of the database.

IMPORTANT **FEEDBACK AND ASYNCHRONOUS METHODS**

Using any of the asynchronous methods available with the SqlDataReader, you can provide feedback (although somewhat limited) to the client application. This enables you to write the application in such a way that the end user can see instantaneous feedback that something is happening, particularly with large result sets. DataReaders have a property called HasRows, which indicates whether data was returned from the query, but there is no way to know the exact number of rows without iterating through the DataReader and counting them. By contrast, the DataAdapter immediately makes the returned record count for each table available upon completion.

EXAM TIP

Although ADO.NET is versatile, powerful, and easy to use, it's the simplest of the choices available. When studying for the exam, you won't have to focus on learning every nuance of every minor member of the System.Data or System.Data.SqlClient namespace. This is a Technical Specialist exam, so it serves to verify that you are familiar with the technology and can implement solutions using it. Although this particular objective is titled "Choose data access technology," you should focus on how you'd accomplish any given task and what benefits each of the items brings to the table. Trying to memorize every item in each namespace is certainly one way to approach this section, but focusing instead on "How do I populate a DataSet containing two related tables using a DataAdapter?" would probably be a much more fruitful endeavor.

Why choose ADO.NET?

So what are the reasons that would influence one to use traditional ADO.NET as a data access technology? What does the exam expect you to know about this choice? You need to be able to identify what makes one technology more appropriate than another in a given setting. You also need to understand how each technology works.

The first reason to choose ADO.NET is consistency. ADO.NET has been around much longer than other options available. Unless it's a relatively new application or an older application that has been updated to use one of the newer alternatives, ADO.NET is already being used to interact with the database.

The next reason is related to the first: stability both in terms of the evolution and quality of the technology. ADO.NET is firmly established and is unlikely to change in any way other than feature additions. Although there have been many enhancements and feature improvements, if you know how to use ADO.NET in version 1.0 of the .NET Framework, you will know how to use ADO.NET in each version up through version 4.5. Because it's been around so long, most bugs and kinks have been fixed.

ADO.NET, although powerful, is an easy library to learn and understand. Once you understand it conceptually, there's not much left that's unknown or not addressed. Because it has

been around so long, there are providers for almost every well-known database, and many lesser-known database vendors have providers available for ADO.NET. There are examples showing how to handle just about any challenge, problem, or issue you would ever run into with ADO.NET.

One last thing to mention is that, even though Windows Azure and cloud storage were not on the list of considerations back when ADO.NET was first designed, you can use ADO.NET against Windows Azure's SQL databases with essentially no difference in coding. In fact, you are encouraged to make the earlier SqlDataAdapter or SqlDataReader tests work against a Windows Azure SQL database by modifying only the connection string and nothing else!

Choosing EF as the data access technology

EF provides the means for a developer to focus on application code, not the underlying "plumbing" code necessary to communicate with a database efficiently and securely.

The origins of EF

Several years ago, Microsoft introduced *Language Integrated Query (LINQ)* into the .NET Framework. LINQ has many benefits, one of which is that it created a new way for .NET developers to interact with data. Several flavors of LINQ were introduced. *LINQ-to-SQL* was one of them. At that time (and it's still largely the case), RDBMS systems and *object oriented programming (OOP)* were the predominant metaphors in the programming community. They were both popular and the primary techniques taught in most computer science curriculums. They had many advantages. OOP provided an intuitive and straightforward way to model real-world problems.

The relational approach for data storage had similar benefits. It has been used since at least the 1970s, and many major vendors provided implementations of this methodology. Most all the popular implementations used an ANSI standard language known as *Structured Query Language (SQL)* that was easy to learn. If you learned it for one database, you could use that knowledge with almost every other well-known implementation out there. SQL was quite powerful, but it lacked many useful constructs (such as loops), so the major vendors typically provided their own flavor in addition to basic support for ANSI SQL. In the case of Microsoft, it was named *Transact SQL* or, as it's commonly known, *T-SQL*.

Although the relational model was powerful and geared for many tasks, there were some areas that it didn't handle well. In most nontrivial applications, developers would find there was a significant gap between the object models they came up with via OOP and the ideal structures they came up with for data storage. This problem is commonly referred to as impedance mismatch, and it initially resulted in a significant amount of required code to deal with it. To help solve this problem, a technique known as *object-relational mapping (ORM, O/RM, or O/R Mapping*) was created. LINQ-to-SQL was one of the first major Microsoft initiatives to build an ORM tool. By that time, there were several other popular ORM tools, some open source and some from private vendors. They all centered on solving the same essential problem.

Compared to the ORM tools of the time, many developers felt LINQ-to-SQL was not powerful and didn't provide the functionality they truly desired. At the same time that LINQ-to-SQL was introduced, Microsoft embarked upon the EF initiative. EF received significant criticism early in its life, but it has matured tremendously over the past few years. Right now, it is powerful and easy to use. At this point, it's also widely accepted as fact that the future of data access with Microsoft is the EF and its approach to solving problems.

The primary benefit of using EF is that it enables developers to manipulate data as domain-specific objects without regard to the underlying structure of the data store. Microsoft has made (and continues to make) a significant investment in the EF, and it's hard to imagine any scenario in the future that doesn't take significant advantage of it.

From a developer's point of view, EF enables developers to work with entities (such as Customers, Accounts, Widgets, or whatever else they are modeling). In EF parlance, this is known as the conceptual model. EF is responsible for mapping these entities and their corresponding properties to the underlying data source.

To understand EF (and what's needed for the exam), you need to know that there are three parts to the EF modeling. Your .NET code works with the conceptual model. You also need to have some notion of the underlying storage mechanism (which, by the way, can change without necessarily affecting the conceptual model). Finally, you should understand how EF handles the mapping between the two.

EF modeling

For the exam and for practical use, it's critical that you understand the three parts of the EF model and what role they play. Because there are only three of them, that's not difficult to accomplish.

The conceptual model is handled via what's known as the *conceptual schema definition language (CSDL)*. In older versions of EF, it existed in a file with a .csdl extension. The data storage aspect is handled through the *store schema definition language (SSDL)*. In older versions of EF, it existed in a file with an .ssdl file extension. The mapping between the CSDL and SSDL is handled via the *mapping specification language (MSL)*. In older versions of EF, it existed in a file with an .msl file extension. In modern versions of EF, the CSDL, MSL, and SSDL all exist in a file with an .edmx file extension. However, even though all three are in a single file, it is important to understand the differences between the three.

Developers should be most concerned with the conceptual model (as they should be); database folk are more concerned with the storage model. It's hard enough to build solid object models without having to know the details and nuances of a given database implementation, which is what DBAs are paid to do. One last thing to mention is that the back-end components can be completely changed without affecting the conceptual model by allowing the changes to be absorbed by the MSL's mapping logic.

Compare this with ADO.NET, discussed in the previous section. If you took any of the samples provided and had to change them to use an Oracle database, there would be major changes necessary to all the code written. In the EF, you'd simply focus on the business objects and let the storage model and mappings handle the change to how the data came from and got back to the database.

Building EF models

The early days of EF were not friendly to the technology. Many people were critical of the lack of tooling provided and the inability to use industry-standard architectural patterns because they were impossible to use with EF. Beginning with version 4.0 (oddly, 4.0 was the second version of EF), Microsoft took these problems seriously. By now, those complaints have been addressed.

There are two basic ways you can use the set of *Entity Data Model (EDM)* tools to create your conceptual model. The first way, called *Database First*, is to build a database (or use an existing one) and then create the conceptual model from it. You can then use these tools to manipulate your conceptual model. You can also work in the opposite direction in a process called *Model First*, building your conceptual model first and then letting the tools build out a database for you. In either case, if you have to make changes to the source or you want to change the source all together, the tools enable you to do this easily.

> ***NOTE*** **CODE FIRST**
>
> **An alternative way to use EF is via a *Code First* technique. This technique enables a developer to create simple classes that represent entities and, when pointing EF to these classes, enables the developer to create a simple data tier that just works. Although you are encouraged to further investigate this technique that uses no .edmx file, the exam does not require that you know how to work with this technique much beyond the fact that it exists. As such, anywhere in this book that discusses EF, you can assume a Model First or Database First approach.**

When you create a new EF project, you create an .edmx file. It's possible to create a project solely from XML files you write yourself, but that would take forever, defeat the purpose for using the EF, and generally be a bad idea. The current toolset includes four primary items that you need to understand:

- The *Entity Model Designer* is the item that creates the .edmx file and enables you to manipulate almost every aspect of the model (create, update, or delete entities), manipulate associations, manipulate and update mappings, and add or modify inheritance relationships.
- The *Entity Data Model Wizard* is the true starting point of building your conceptual model. It enables you to use an existing data store instance.

- The *Create Database Wizard* enables you to do the exact opposite of the previous item. Instead of starting with a database, it enables you to fully build and manipulate your conceptual model, and it takes care of building the actual database based on the conceptual model.
- The *Update Model Wizard* is the last of the tools, and it does exactly what you'd expect it to. After your model is built, it enables you to fully modify every aspect of the conceptual model. It can let you do the same for both the storage model and the mappings that are defined between them.

There's one other tool that's worth mentioning, although it's generally not what developers use to interact with the EF. It's known as the *EDM Generator* and is a command-line utility that was one of the first items built when the EF was being developed. Like the combination of the wizard-based tools, it enables you to generate a conceptual model, validate a model after it is built, generate the actual C# or VB.NET classes that are based off of the conceptual model, and also create the code file that contains model views. Although it can't hurt to know the details of how this tool works, the important aspects for the exam focus on each of the primary components that go into an EDM, so it is important to understand what each of those are and what they do.

Building an EF Model using the Entity Data Model Wizard

This section shows you how to use the tools to build a simple model against the TestDB created in the beginning of Chapter 1. You can alternatively manually create your models and use those models to generate your database if you alter step 3 and choose Empty Model instead.

However, before you begin, make sure that your TestDB is ready to go, and you're familiar with how to connect to it. One way is to ensure that the tests back in the ADO.NET section pass. Another way is to ensure that you can successfully connect via *SQL Server Management Studio (SSMS)*. For the included screen shots, the EF model is added to the existing MySimpleTests project.

1. First, right-click on your Project in Solution Explorer and add a New Item.
2. In the Add New Item dialog box, select Visual C# Items → Data on the left and ADO.NET Entity Data Model in the middle (don't let the name of this file type throw you off because it does include "ADO.NET" in the name). Name this **MyModel.edmx** and click Add (see Figure 1-2).

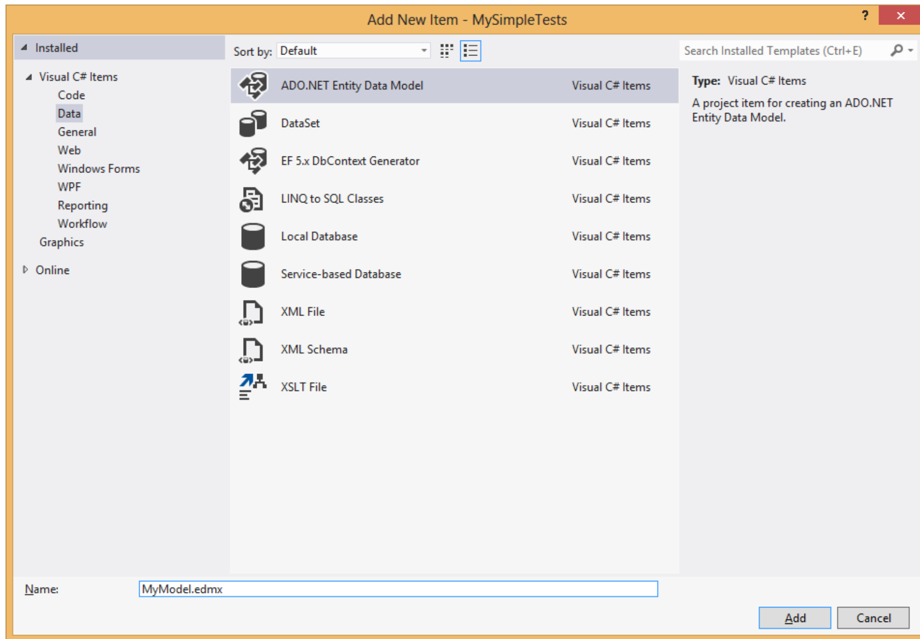

FIGURE 1-2 ADO.NET Entity Data Model Wizard dialog box

3. In the Entity Data Model Wizard dialog box, select Generate From Database and click Next.
4. Next, the Entity Data Model Wizard requires that you connect to your database. Use the New Connection button to connect and test your connection. After you're back to the Entity Data Model Wizard dialog box, ensure that the check box to save your connection settings is selected and name it **TestEntities** (see Figure 1-3).

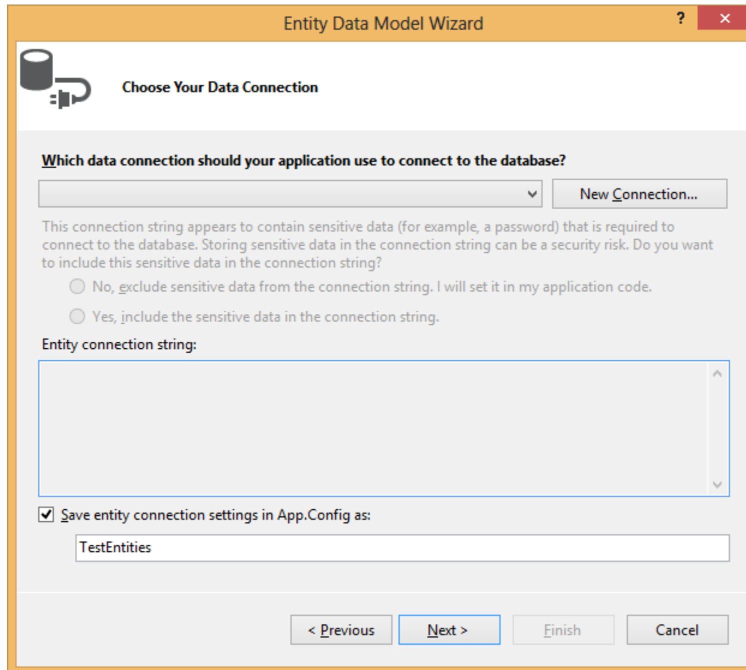

FIGURE 1-3 Choose Your Data Connection dialog box

5. In the next screen of the Entity Data Model Wizard, select all the tables and select both check boxes under the database objects. Finally, for the namespace, call it **TestModel** and click Finish (see Figure 1-4).

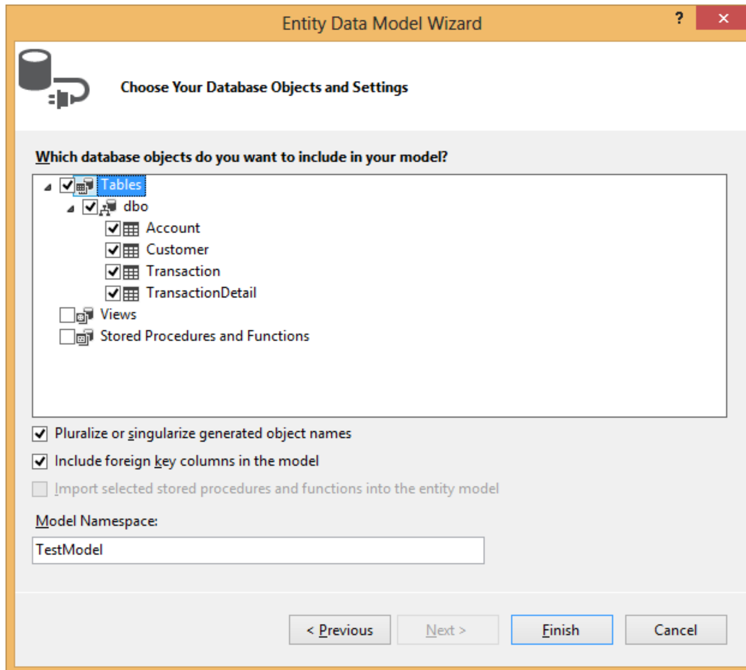

FIGURE 1-4 Choose Your Database Objects And Settings dialog box

6. You should now see the Entity Model Designer in a view that looks similar to an entity relationship diagram shown in Figure 1-5.

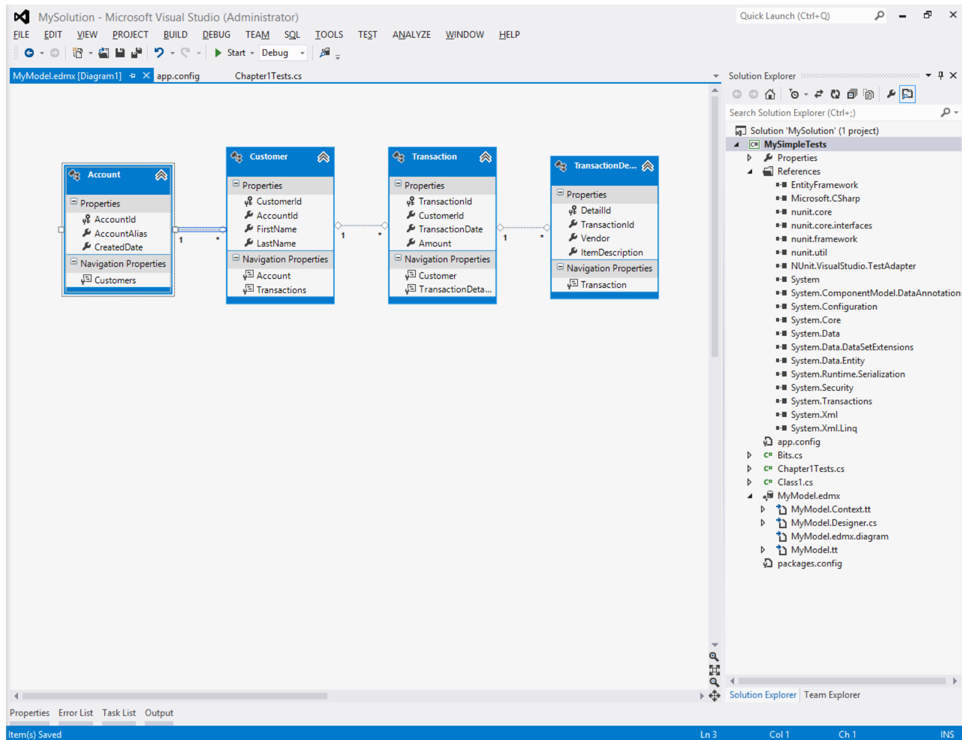

FIGURE 1-5 Entity Model view

After generating your EF models, you now have a fully functioning data tier for simple consumption! Quickly test this to investigate everything you just created. See the code for a quick test of your new EF data tier:

Entity Framework test

```
[TestCase(3)]
public static void GetCustomerById(int customerId)
{
    // ARRANGE
    TestEntities database = new TestEntities();

    // ACT
    Customer result = database.Customers.SingleOrDefault(cust => cust.CustomerId ==
customerId);

    // ASSERT
    Assert.That(result, Is.Not.Null, "Expected a value. Null here indicates no record
with this ID.");
    Assert.That(result.CustomerId, Is.EqualTo(customerId), "Uh oh!");
}
```

There are a few things to note about what happens in this test. First, the complexity of the code to consume EF is completely different compared with the prior ADO.NET tests! Second,

this test runs in approximately 4 ms compared with the 2–3 ms for ADO.NET. The difference here isn't so much 50–100 percent, but rather 1–2 ms for such a simple query. Finally, the query that runs against SQL Server in this case is substantially different from what was run with the ADO.NET queries for two reasons: EF does some ugly (although optimized) aliasing of columns, tables, and parameters; and this performs a SELECT TOP (2) to enforce the constraint from the use of the Linq SingleOrDefault command.

> ***MORE INFO*** **SINGLE VERSUS FIRST IN LINQ**
>
> **When using LINQ, if you have a collection and you want to return a single item from it, you have two obvious options if you ignore nulls (four if you want to handle nulls). The First function effectively says, "Give me the first one of the collection." The Single function, however, says, "Give me the single item that is in the collection and throw an exception if there are more or fewer than one item." In both cases, the xxxOrDefault handles the case when the collection is empty by returning a null value. A bad programming habit that many developers have is to overuse the First function when the Single function is the appropriate choice. In the previous test, Single is the appropriate function to use because you don't want to allow for the possibility of more than one Customer with the same ID; if that happens, something bad is going on!**

As shown in Figure 1-6, there's a lot behind this .edmx file you created. There are two things of critical importance and two things that are mostly ignorable. For now, ignore the MyModel.Designer.cs file, which is not currently used, and ignore the MyModel.edmx.diagram file, which is used only for saving the layout of tables in the visual designer.

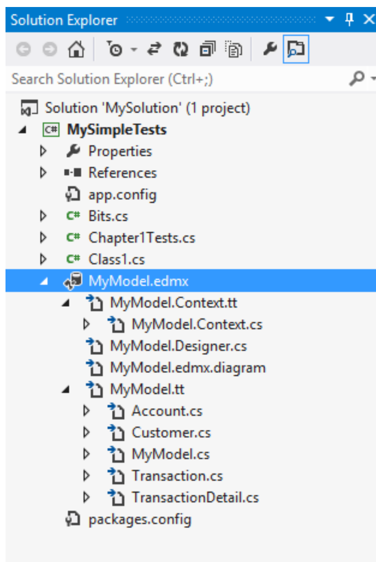

FIGURE 1-6 EF-generated files

MORE INFO T4 CODE GENERATION

T4 text template files can often be identified by the .tt extension. *T4* is the templating and code generation engine that EF uses to generate code so you don't have to manage it yourself. It's useful if you know how to automatically generate code in bulk based on known variables. In this case, your known variables are your database structure and, more importantly, your conceptual model. Because you created the various models in your .edmx file, you now have a handy definition file for your conceptual model.

First, look at the MyModel.tt file and then its Customer.cs file. Double-click the MyModel.tt file and you'll see the contents of this T4 text template. This template generates simple classes that represent the records in your tables in your database. Now open the Customer.cs file. The only two pieces of this file that might be new to you are the ICollection and HashSet types. These are simple collection types that are unrelated to EF, databases, or anything else. ICollection<T> is a simple interface that represents a collection; nothing special here. A HashSet<T> is similar to a generic List<T>, but is optimized for fast lookups (via hashtables, as the name implies) at the cost of losing order.

T4-Generated Customer.cs

```
public partial class Customer
{
    public Customer()
    {
        this.Transactions = new HashSet<Transaction>();
    }

    public int CustomerId { get; set; }
    public int AccountId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public virtual Account Account { get; set; }
    public virtual ICollection<Transaction> Transactions { get; set; }
}
```

Next, look at the file MyModel.Context.tt generated: MyModel.Context.cs. There are three important things to see in this file. First, the TestEntities class inherits the DbContext class. This class can be thought of as the EF runtime that does all the magic work. The DbContext API was introduced with EF 4.1, and Microsoft has an excellent series of documentation on how to work with this API at *http://msdn.microsoft.com/en-us/data/gg192989.aspx*.

Inheriting DbContext enables it to talk to a database when coupled with the .edmx file and the connection string in the config file. Looking at this example, notice that the parameter passes to the base constructor. This means that it depends on the config file having a properly configured EF connection string named TestEntities in the config file. Take a look at it and notice how this connection string differs from the one you used for the ADO.NET tests. Also

notice the DbSet collections. These collections are what enable us to easily work with each table in the database as if it were simply a .NET collection. Chapter 2, "Querying and manipulating data by using the Entity Framework," investigates this in more detail.

T4-Generated MyModel.Context.cs

```
public partial class TestEntities : DbContext
{
    public TestEntities()
        : base("name=TestEntities")
    {
    }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        throw new UnintentionalCodeFirstException();
    }

    public DbSet<Account> Accounts { get; set; }
    public DbSet<Customer> Customers { get; set; }
    public DbSet<Transaction> Transactions { get; set; }
    public DbSet<TransactionDetail> TransactionDetails { get; set; }
}
```

Entity Data Model Designer

When you double-click the .edmx file and are given the ERD-like view, you invoke the *EDM Designer*. This is the primary tool for manipulating your models, whether you're manually creating a new one, updating your existing models based on changes in the schema of your data, or choosing to manually tweak your conceptual model without changes to your storage model.

When you have the Designer open, there is a window that is useful to have at times that can be opened by clicking View → Other Windows → Entity Data Model Browser. This is especially useful as your .edmx begins to cover a large number of tables and you have difficulty locating a particular one.

If you want to manage your models in the Designer, typically you'd just right-click anywhere inside the design canvas of the .edmx model file you're working with or on a specific entity or property that you want to manage. If you want to update your models based on data schema changes, it can be done with the Update Model Wizard by right-clicking in the design canvas and selecting Update Model From Database. In the Update Model Wizard, you can choose to add new tables or simply click Finish for it to pull in schema changes to existing tables.

The options that might raise some questions are likely to be the Inheritance and Complex Type options. Otherwise, editing the conceptual model is straightforward. Inheritance is a way to create entities with OOP class hierarchies that is primarily used in Code First and Model First (but not Database First) scenarios. With Inheritance, you can have both an Employee and

Customer type that inherits a Person type. Although an Employee might have a Salary field, a Customer certainly wouldn't. Behind-the-scenes in your data store, this can be represented in a couple different ways.

The default mapping strategy, called *Table per Hierarchy (TPH)*, is the most straightforward way to do this. It creates a single table for all objects in an inheritance hierarchy, and it simply has nullable columns for fields that aren't common across all types. It also adds a Discriminator column so EF can keep track of the type of each individual record. This strategy is often the best balance of tradeoffs available because it provides the best performance. The primary disadvantage is that your data is slightly denormalized.

The prevailing alternative strategy, called *Table per Type (TPT)*, creates a table for your base type that has all common fields in it, a table for each child type that stores the additional fields, as well as an ID to the base type's record that stores the common fields. The multiple inheriting types' tables are linked to one another via a foreign key that has a shared primary key value. In this case, to get all the data for a single entity, EF must perform a join across multiple tables. The primary advantage is that your data is properly normalized. The primary disadvantage is that your performance suffers.

Your other mapping options include *Table per Concrete Type (TPC)* and *Mixed Inheritance*. These two options are not currently supported in the EDM Designer, although the EF runtime does support them. The exam will likely not cover these mapping options, nor are they usually your most practical choice to use because of the limited tooling support for them.

A *Complex Type* is a logical designation for a common group of fields on multiple entities. When you have a Complex Type, your conceptual model has an object to work with for easier usability of these repeated groups of fields (an example is a date range). Many tables in many databases have a StartDate field and a StopDate field. You could take these two fields, identify them as being parts of a Complex Type called a DateRange, and then your conceptual model could specify when an entity has a Date Range instead of arbitrary date fields.

The only other major item is the actual editing of the conceptual entity items. If you select any given entity on the conceptual model canvas, right-click it, and select Add New, you'll see a menu with the following items:

- Scalar Property
- Navigation Property
- Complex Property
- Association
- Inheritance
- Function Import

From there, the remaining features are self-explanatory. Following is a brief description of each:

- **Table Mapping** enables you to change the relationship to the underlying table. For instance, you can add fields, remove fields, or change fields from the specific property originally mapped to.
- **Stored Procedure Mapping** is based on the Stored Procedures defined in the database. It enables you specify an Insert, Update, and Delete function. Select queries are already handled by the language semantics of LINQ.
- **Update Model from Database** is the same behavior described previously in the canvas-based designer.
- **Generate Database from Model** is the same behavior described previously in the canvas-based designer.
- **Validate** is the same behavior described previously in the canvas-based designer that ensures the validity of the .edmx file.

Using the generated items

So far, everything has involved the .edmx files and their respective components. It's critical to understand each of the pieces and why they matter when designing an application, but understanding how to generate a model is only one part of the equation. A tremendous amount of time is spent developing and maintaining business objects and their interaction with the database, so having this handled for you is beneficial. But just as important is to understand what you do with everything after it is generated.

In the simplest scenario, the conceptual model provides you with classes that map directly to the underlying tables in your RDBMS structure (unless, of course, you specify otherwise). These classes are actual C# or VB.NET code, and you can work with them just like any other class you create. To understand things and prepare for the exam, I highly recommend that you generate a few different .edmx files (try both Database First and Model First) and examine them closely.

At the top level, the primary object you'll work with is an ObjectContext or DbContext object. Choose a name for the context early on in the wizard. It can be changed just like everything else, but the important takeaway for the exam is that this is the primary item from which everything else flows. If you do want to change the name, however, you simply have focus in the Designer for the .edmx file and set the Entity Container Name value in the Properties window.

ObjectContext versus DbContext

Older versions of EF did not have DbContext. Instead, ObjectContext was the equivalent class that funneled most of the functionality to the developer. Modern versions of EF still support the ObjectContext object, and you can even consume modern EF in much the same way as the older versions. Suffice it to say, if you're beginning a new project, DbContext is where you should look to for your EF needs. However, both flavors of the API are important from a real-world legacy code perspective as well as a testing perspective for this exam. In other

words, be prepared to answer questions about both ways of using EF. Fortunately, many of the concepts from the one flavor are directly applicable to the other flavor.

The steps for creating an .edmx are intended for creating a DbContext usage scenario. If you want to work in an ObjectContext scenario, you have three primary options. The intimate details of these approaches are beyond this book; you are encouraged to investigate these approaches:

- **Legacy approach** Follow similar steps using Visual Studio 2008 or 2010 to create an .edmx file and its corresponding ObjectContext and entities.
- **Downgrading your entities** Take the .edmx file that was generated, open the Properties window, and set focus in the Designer's canvas. From here, change the Code Generation Strategy from None to Default in the Properties window. Finally, delete the two .tt files listed as children to your .edmx file in the Solution Explorer window.
- **Hybrid approach** Get the ObjectContext (via a nonobvious approach) from the DbContext and work with the ObjectContext directly. Note that even with modern versions of EF, in some rare and advanced scenarios this is still required:

Hybrid approach

```
public static ObjectContext ConvertContext(DbContext db)
{
    return ((IObjectContextAdapter)db).ObjectContext;
}

[TestCase(3)]
public static void GetCustomerByIdOnObjectContext(int customerId)
{
    // ARRANGE
    TestEntities database = new TestEntities();
    ObjectContext context = ConvertContext(database);

    // ACT
    ObjectSet<Customer> customers = context.CreateObjectSet<Customer>("Customers");
    Customer result = customers.SingleOrDefault(cust => cust.CustomerId == customerId);
    //Customer result = database.Customers.SingleOrDefault(cust => cust.CustomerId ==
customerId);

    // ASSERT
    Assert.That(result, Is.Not.Null, "Expected a value. Null here indicates no record
with this ID.");
    Assert.That(result.CustomerId, Is.EqualTo(customerId), "Uh oh!");
}
```

As you can tell from this test with the commented line of code, basic consumption of a DbContext is quite similar to that of an ObjectContext. In fact, most T4 templated instances of ObjectContexts already have properties pregenerated for you, so you can simply access the equivalent ObjectSet<Customer> collection directly off of the generated class that derives ObjectContext. This simple consumption truly is identical to that of a the generated DbContext-derived class and its DbSet collection.

ObjectContext management

Two important things happen when an ObjectContext constructor is called. The generated context inherits several items from the ObjectContext base class, including a property known as ContextOptions and an event named OnContextCreated. The ContextOptions class has five primary properties you can manipulate:

- LazyLoadingEnabled
- ProxyCreationEnabled
- UseConsistentNullReferenceBehavior
- UseCSharpNullComparisonBehavior
- UseLegacyPreserveChangesBehavior

The LazyLoadingEnabled property is set when any instance of the context is instantiated and is used to enable lazy loading. If left unspecified, the default setting is true. Lazy loading enables entities to be loaded on-demand without thought by the developer. Although this can be handy in some scenarios, this behavior can have very serious performance implications depending on how relationships are set up. A good deal of thought should be taken into consideration regarding this enabling or disabling lazy loading. Feature development and even application architecture might change one way or another based on the use of this feature. And some architectures make it necessary to disable this feature.

In EF, lazy loading is triggered based on a Navigation Property being accessed. By simply referencing a navigation property, that entity is then loaded. As an example, let's take the Customer class and the related Transaction class. If LazyLoadingEnabled is set to true, and you load one customer, you'll just get back data just for that customer record. But if you later access a Navigation Property within the same ObjectContext, another roundtrip to the database will be made to fetch that data. If you loop through a collection of entities (in this case, the Transactions), an individual roundtrip is made to the database for each entity. If you have LazyLoadingEnabled set to false, you have two options. You can either use explicit lazy loading with the ObjectContext's LoadProperty() method or you can use eager loading with the ObjectSet's Include() method. With explicit lazy loading, you must write code to explicitly perform this additional roundtrip to the database to conditionally load the data. With eager loading, you must specify up front what related data you want loaded. Although explicit lazy loading, like regular lazy loading, can reduce the amount of data flowing back and forth, it's a chatty pattern, whereas eager loading is a chunky pattern.

***NOTE* CHATTY VERSUS CHUNKY**

Assume it takes 100 ms per MB of data. In just transfer times alone, that puts you at 2.5 seconds versus 1 second transferring data. However, assume there is also a 100 ms delay for every round trip (quite common on the Internet and is not too terribly uncommon in high-availability environments). In addition to data transfer times, you must also add 100 ms to the 25 MB transfer, but a huge 10 seconds to the 10 MB transfer times. So in the end, it would take only 2.6 seconds for 25 MB, but 11 seconds for 10 MB, simply because you chose to be chatty instead of chunky with the architectural decision! Now if the transfer times were the same but the latency was only 1 ms (back to the two VMs on one host), you would be looking at 2.501 seconds versus 1.1 seconds, and the chattier of the two options would have been the better choice!

Of the five ContextOption properties, LazyLoadingEnabled is by far the most important one with the most serious implications for your application. Although it is inappropriate to refer to the other properties as unimportant, lazy loading can very quickly make or break your application.

The ProxyCreationEnabled property is one of the other properties of the ObjectContextOptions class that simply determines whether proxy objects should be created for custom data classes that are persistence-ignorant, such as *plain old common object (POCO)* entities (POCO entities are covered more in Chapter 2). It defaults to true and, unless you have some specific reason to set it to false, it generally isn't something you need to be overly concerned about.

This property is available only in newer versions of EF. A very common problem with this property occurs if you initially target .NET Framework 4.5 with your application code, reference this property, and then later drop down to .NET 4.0 where this property does not exist.

This functionality that this property controls is a little confusing if you disable lazy loading and load a Customer object from the TestDB database. If you inspect the AccountId, you can see a value of the parent record. But if you inspect the Account navigation property, it is null because it has not yet been loaded. Now with the UseConsistentNullReferenceBehavior property set to false, you can then attempt to set the Account navigation property to null, and nothing will happen when you try to save it. However, if this setting is set to true, the act of setting the Account navigation property to null attempts to sever the relationship between the two records upon saving (which results in an error because the AccountId is not a nullable field). If, on the other hand, you had the Customer object loaded and its Account property was also loaded (either by eager loading or lazy loading), the UseConsistentNullReferenceBehavior setting has no effect on setting the Account property to null.

As the name implies, C# NullComparison behavior is enabled when set to true; otherwise, it is not enabled. The main implication of this property is that it changes queries that require null comparisons. For example, take a look at a test and see what happens. To set the data up for this test, ensure that your Account table has three records in it: one with an AccountAlias set to Home, one with an AccountAlias set to Work, and one with AccountAlias set to null. You can tweak the second parameter of the test to other numbers if you want to work with a different number of records:

Null comparison behavior

```
[TestCase(true, 2)]
[TestCase(false, 1)]
public static void GetAccountsByAliasName(bool useCSharpNullBehavior, int recordsToFind)
{
    // ARRANGE
    TestEntities database = new TestEntities();
    ObjectContext context = ConvertContext(database);
    ObjectSet<Account> accounts = context.CreateObjectSet<Account>("Accounts");

    // ACT
    context.ContextOptions.UseCSharpNullComparisonBehavior = useCSharpNullBehavior;
    int result = accounts.Count(acc => acc.AccountAlias != "Home");

    // ASSERT
    Assert.That(result, Is.EqualTo(recordsToFind), "Uh oh!");
}
```

Behind the scenes, when this setting is enabled, EF automatically tweaks your query to also include cases in which AccountAlias is set to null. In other words, depending on whether it is enabled or disabled, you might see the two following queries hitting your database:

```
SELECT COUNT(*) FROM Account WHERE AccountAlias <> 'Home' – Disabled
SELECT COUNT(*) FROM Account WHERE AccountAlias <> 'Home' OR AccountAlias IS NULL --
Enabled
```

This property can be set to true or false and is quite confusing. This setting causes EF to use either the .NET Framework 4.0 (and newer) functionality for MergeOption.PreserveChanges or use the .NET Framework 3.5 SP1 version when merging two entities on a single context. Setting this to true gives you some control over how to resolve Optimistic Concurrency Exceptions while attaching an entity to the ObjectContext when another copy of that same entity already exists on the context. Starting with.NET Framework version 4.0, when you attach an entity to the context, the EF compares the current local values of unmodified properties against those already loaded in the context. If they are the same, nothing happens. If they are different, the state of the property is set to Modified. The legacy option changes this behavior. When the entity is attached, the values of unmodified properties are not copied over; instead, the context's version of those properties is kept. This might be desirable, depending on how you want to handle concurrency issues in your application. Dealing with concurrency issues and all the implications that come with it is well beyond the scope of both the exam and this book.

ObjectContext entities

Understanding the structure of ObjectContext is critical for the exam. In addition, you need to understand the structure of the Entity classes that are generated for it. Thanks to the appropriate T4 templates, each entity that you define in the conceptual model gets a class generated to accompany it. There are a few options for the types of entities that can be generated. In EF 4.0, the default base class for entities was EntityObject. Two other types of entities could also be created: POCO entities (no required base class for these) and Self-Tracking entities (STEs) (no base class, but they implemented IObjectWithChangeTracker and INotifyPropertyChanged). Chapter 2 covers POCO entities and STE in more detail, but very little knowledge of them is required for the exam.

The ObjectContext entities of significant interest inherit from the EntityObject class. The class declaration for any given entity resembles the following:

```
public partial class Customer : EntityObject
{
    /* ... */
}
```

EXAM TIP

As noted earlier, you need to pay attention to the derived class that the given context is derived from because it is likely to be included in the exam. The same is the case with entities that derive from the EntityObject base class. These entities have several attributes decorating the class definition that you should be familiar with. There are also several similar distinctions surrounding properties of each entity class (or *attributes*, as they are called in the conceptual model).

Next, each entity will be decorated with three specific attributes:

- EdmEntityTypeAttribute
- SerializableAttribute
- DataContractAttribute

So the full declaration looks something like the following:

```
[EdmEntityTypeAttribute(NamespaceName = "MyNameSpace", Name = "Customer")]
[Serializable()]
[DataContractAttribute(IsReference = true)]
public partial class Customer : EntityObject
{
    /* ... */
}
```

The Entity class needs to be serializable. If you want to persist the object locally, pass it between tiers, or do much else with it, it has to be serializable. Because ObjectContext's incarnations of the EF were created to work in conjunction with WCF and WCF Data Services, the