THE EXPERT'S VOICE® IN .NET

Entity Framework 4.0 Recipes

A Problem-Solution Approach

Ready-made solutions for putting Microsoft® Entity Framework 4.0 to work in your own applications

Larry Tenny and Zeeshan Hirani

Apress®

Table of Contents

Copyright 1
Dedication 2
About the Authors 29
About the Technical Reviewers 30
Acknowledgments 31
Preface 32
Chapter 1. Getting Started With Entity Framework 38
Section 1.1. A Brief Tour of the Entity Framework World 39
Section 1.2. Using Entity Framework 42
Chapter 2. Entity Data Modeling Fundamentals 46
Section 2.1. Creating a Simple Model 46
Section 2.2. Creating a Model from an Existing Database 53
Section 2.3. Modeling a Many-to-Many Relationship with No Payload 59
Section 2.4. Modeling a Many-to-Many Relationship with a Payload 63
Section 2.5. Modeling a Self-Referencing Relationship 66
Section 2.6. Splitting an Entity Across Multiple Tables 70
Section 2.7. Splitting a Table Across Multiple Entities 74
Section 2.8. Modeling Table per Type Inheritance 79
Section 2.9. Using Conditions to Filter an ObjectSet 83
Section 2.10. Modeling Table per Hierarchy Inheritance 86
Section 2.11. Modeling Is-a and Has-a Relationships Between Two Entities 91
Section 2.12. Creating, Modifying, and Mapping Complex Types 94
Chapter 3. Querying an Entity Data Model 100
Section 3.1. Executing an SQL Statement 100
Section 3.2. Returning Objects from a SQL Statement 103
Section 3.3. Returning Objects from an Entity SQL Statement 105
Section 3.4. Specifying Fully Qualified Names in Entity SQL 109
Section 3.5. Finding a Master that Has Detail in a Master-Detail Relationship 111
Section 3.6. Setting Default Values in a Query 114
Section 3.7. Returning Multiple Result Sets From a Stored Procedure 117
Section 3.8. Comparing Against a List of Values 119
Section 3.9. Building and Executing a Query Against an ObjectSet<T> 122
Section 3.10. Returning a Primitive Type From a Query 124
Section 3.11. Filtering Related Entities 126
Section 3.12. Applying a Left Outer Join 130
Section 3.13. Ordering by Derived Types 133
Section 3.14. Paging and Filtering 135
Section 3.15. Grouping by Date 138
Section 3.16. Flattening Query Results 140
Section 3.17. Grouping by Multiple Properties 142
Section 3.18. Using Bitwise Operators in a Filter 145
Section 3.19. Joining on Multiple Columns 148
Chapter 4. Using Entity Framework in ASP.NET 152
Section 4.1. Building a Search Query 152
Section 4.2. Building CRUD Operations in an ASP.NET Web Page 156
Section 4.3. Executing Business Logic When Changes Are Saved 161
Section 4.4. Loading Related Entities 164
Section 4.5. Searching with QueryExtender 166
Section 4.6. Retrieving a Derived Type Using an EntityDataSource Control 173
Section 4.7. Filtering with ASP.NET's URL Routing 176
Section 4.8. Building CRUD Operations with an ObjectDataSource Control 180
Section 4.9. Using Entity Framework With MVC 186
Chapter 5. Loading Entities and Navigation Properties 192
Section 5.1. Loading Related Entities 192
Section 5.2. Loading a Complete Object Graph 197
Section 5.3. Loading Navigation Properties on Derived Types 199
Section 5.4. Using Include() with Other LINQ Query Operators 202
Section 5.5. Deferred Loading of Related Entities 204
Section 5.6. Filtering and Ordering Related Entities 206
Section 5.7. Executing Aggregate Operations on Related Entities 209
Section 5.8. Testing Whether an Entity Reference or Entity Collection Is Loaded 211
Section 5.9. Loading Related Entities Explicitly 213
Section 5.10. Filtering an Eagerly Loaded Entity Collection 217
Section 5.11. Using Relationship Span 219
Section 5.12. Modifying Foreign Key Associations 221

Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Chapter 6. Beyond the Basics with Modeling and Inheritance..... 226
Section 6.1. Retrieving the Link Table in a Many-to-Many Association..... 226
Section 6.2. Exposing a Link Table as an Entity..... 229
Section 6.3. Modeling a Many-to-Many, Self-Referencing Relationship..... 233
Section 6.4. Modeling a Self-Referencing Relationship Using Table per Hierarchy Inheritance..... 237
Section 6.5. Modeling a Self-Referencing Relationship and Retrieving a Complete Hierarchy..... 241
Section 6.6. Mapping Null Conditions in Derived Entities..... 245
Section 6.7. Modeling Table per Type Inheritance Using a Non-Primary Key Column..... 248
Section 6.8. Modeling Nested Table per Hierarchy Inheritance..... 253
Section 6.9. Limiting the Values Assigned to a Foreign Key..... 257
Section 6.10. Applying Conditions in Table per Type Inheritance..... 261
Section 6.11. Creating a Filter on Multiple Criteria..... 264
Section 6.12. Using Complex Conditions with Table per Hierarchy Inheritance..... 269
Section 6.13. Modeling Table per Concrete Type Inheritance..... 275
Section 6.14. Applying Conditions on a Base Entity..... 279
Section 6.15. Creating Independent and Foreign Key Associations..... 283
Section 6.16. Changing an Independent Association into a Foreign Key Association..... 284
Chapter 7. Working with Object Services..... 288
Section 7.1. Dynamically Building a Connection String..... 288
Section 7.2. Reading a Model from a Database..... 290
Section 7.3. Deploying a Model..... 294
Section 7.4. Using the Pluralization Service..... 295
Section 7.5. Retrieving Entities from the Object State Manager..... 298
Section 7.6. Generating a Model from the Command Line..... 300
Section 7.7. Working with Dependent Entities in an Identifying Relationship..... 301
Section 7.8. Inserting Entities Using an Object Context..... 304
Chapter 8. Plain Old CLR Objects..... 308
Section 8.1. Using POCO..... 308
Section 8.2. Loading Related Entities With POCO..... 313
Section 8.3. Lazy Loading With POCO..... 316
Section 8.4. POCO With Complex Type Properties..... 320
Section 8.5. Notifying Entity Framework About Object Changes..... 323
Section 8.6. Retrieving the Original (POCO) Object..... 326
Section 8.7. Manually Synchronizing the Object Graph and the Object State Manager..... 329
Section 8.8. Testing Domain Objects..... 333
Section 8.9. Testing a Repository Against a Database..... 342
Chapter 9. Using the Entity Framework in N-Tier Applications..... 348
Section 9.1. Deleting an Entity When Disconnected..... 348
Section 9.2. Managing Concurrency When Disconnected..... 352
Section 9.3. Finding Out What Has Changed..... 356
Section 9.4. Using POCO With WCF..... 360
Section 9.5. Using Self-Tracking Entities With WCF..... 366
Section 9.6. Validating Self-Tracking Entities..... 371
Section 9.7. Using Self-Tracking Entities on the Server Side..... 375
Section 9.8. Serializing Proxies in a WCF Service..... 382
Section 9.9. Serializing Self-Tracking Entities in the ViewState..... 386
Section 9.10. Fixing Duplicate References on a WCF Client..... 391
Chapter 10. Stored Procedures..... 396
Section 10.1. Returning an Entity Collection..... 396
Section 10.2. Returning Output Parameters..... 399
Section 10.3. Returning a Scalar Value Result Set..... 402
Section 10.4. Returning a Complex Type from a Stored Procedure..... 404
Section 10.5. Defining a Custom Function in the Storage Model..... 407
Section 10.6. Populating Entities in a Table per Type Inheritance Model..... 410
Section 10.7. Populating Entities in a Table per Hierarchy Inheritance Model..... 413
Section 10.8. Mapping the Insert, Update, and Delete Actions to Stored Procedures..... 416
Section 10.9. Using Stored Procedures for the Insert and Delete Actions in a Many-to-Many Association..... 419
Section 10.10. Mapping the Insert, Update, and Delete Actions to Stored Procedures for Table per Hierarchy Inheritance..... 424
Chapter 11. Functions..... 430
Section 11.1. Returning a Scalar Value from a Model Defined Function..... 430
Section 11.2. Filtering an Entity Collection Using a Model Defined Function..... 434
Section 11.3. Returning a Computed Column from a Model Defined Function..... 438
Section 11.4. Calling a Model Defined Function from a Model Defined Function..... 441
Section 11.5. Returning an Anonymous Type From a Model Defined Function..... 445
Section 11.6. Returning a Complex Type From a Model Defined Function..... 449
Section 11.7. Returning a Collection of Entity References From a Model Defined Function..... 452
Section 11.8. Using Canonical Functions in eSQL..... 455
Section 11.9. Using Canonical Functions in LINQ..... 456
Section 11.10. Calling Database Functions in eSQL..... 459
Section 11.11. Calling Database Functions in LINQ..... 461
Section 11.12. Defining Built-in Functions..... 462

Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Chapter 12. Customizing Entity Framework Objects..... 466
Section 12.1. Executing Code When SaveChanges() Is Called..... 466
Section 12.2. Validating Property Changes..... 469
Section 12.3. Logging Database Connections..... 472
Section 12.4. Recalculating a Property Value When an Entity Collection Changes..... 474
Section 12.5. Automatically Deleting Related Entities..... 477
Section 12.6. Deleting All Related Entities..... 480
Section 12.7. Assigning Default Values..... 484
Section 12.8. Retrieving the Original Value of a Property..... 488
Section 12.9. Retrieving the Original Association for Independent Associations..... 491
Section 12.10. Retrieving XML..... 494
Section 12.11. Applying Server-Generated Values to Properties..... 497
Section 12.12. Validating Entities on SavingChanges..... 501
Chapter 13. Improving Performance..... 508
Section 13.1. Optimizing Queries in a Table per Type Inheritance Model..... 508
Section 13.2. Retrieving a Single Entity Using an Entity Key..... 510
Section 13.3. Retrieving Entities for Read Only..... 512
Section 13.4. Improving the Startup Time..... 514
Section 13.5. Efficiently Building a Search Query..... 516
Section 13.6. Making Change Tracking with POCO Faster..... 519
Section 13.7. Compiling LINQ Queries..... 522
Section 13.8. Returning Partially Filled Entities..... 526
Section 13.9. Moving an Expensive Property to Another Entity..... 528
Section 13.10. Avoiding Include..... 532
Section 13.11. Improving QueryView Performance..... 534
Section 13.12. Generating Proxies Explicitly..... 537
Section 13.13. Preventing the Update of All Columns in Self-Tracking Entities..... 540
Chapter 14. Concurrency..... 546
Section 14.1. Applying Optimistic Concurrency..... 546
Section 14.2. Managing Concurrency When Using Stored Procedures..... 549
Section 14.3. Reading Uncommitted Data..... 553
Section 14.4. Implementing the "Last Record Wins" Strategy..... 555
Section 14.5. Getting Affected Rows from a Stored Procedure..... 557
Section 14.6. Optimistic Concurrency with Table Per Type Inheritance..... 561
Section 14.7. Generating a Timestamp Column with Model First..... 564
Chapter 15. Advanced Modeling..... 566
Section 15.1. Creating an Association on a Derived Entity..... 566
Section 15.2. Mapping an Entity to Customized Parts of One or More Tables..... 569
Section 15.3. Creating Conditional Associations..... 573
Section 15.4. Fabricating Additional Inheritance Hierarchies..... 579
Section 15.5. Sharing Audit Fields Across Multiple Entities..... 584
Section 15.6. Modeling a Many-to-Many Relationship with Payload..... 590
Section 15.7. Mapping a Foreign Key Column to Multiple Associations..... 593
Section 15.8. Using Inheritance to Map a Foreign Key Column to Multiple Associations..... 601
Section 15.9. Creating Read-only and Computed Properties..... 605
Section 15.10. Mapping an Entity to Multiple Tables..... 613
Section 15.11. Mapping an Entity to Multiple Entity Sets (MEST)..... 615
Section 15.12. Extending Table per Type with Table per Hierarchy..... 622
bvdindexIndex..... 628

Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach

Copyright © 2010 by Larry Tenny and Zeeshan Hirani

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

ISBN-13 (pbk): 978-1-4302-2703-8

ISBN-13 (electronic): 978-1-4302-2704-5

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

Trademarked names may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, we use the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

Publisher and President: Paul Manning
Lead Editor: Jonathan Gennick
Technical Reviewers: David Annesley-DeWinter, Brian Swan
Editorial Board: Clay Andres, Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Jonathan Gennick, Jonathan Hassell, Michelle Lowman, Matthew Moodie, Duncan Parkes, Jeffrey Pepper, Frank Pohlmann, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Matt Wade, Tom Welsh
Coordinating Editor: Mary Tobin
Copy Editor: Nancy Sixsmith
Compositor: Bytheway Publishing Services
Indexer: Toma Mulligan
Artist: April Milne
Cover Designer: Anna Ishchenko

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 `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/info/bulksales`.

The information in this book is distributed on an “as is” basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor Apress shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

You can download the examples from the book’s catalog page: `http://apress.com/book/view/1430227036`. Look for the “Source Code” link underneath the cover image. You will need to answer questions pertaining to this book in order to successfully download the code.

Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

To the most important people in my life, my wife and kids.

– Larry

I would like to dedicate this book to my parents for encouraging me and supporting me always. Special thanks to my Dad for his great love and support in everything.

– Zeeshan

Licensed by
Matt Allen
2175280

Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Contents at a Glance

- Contents at a Glance iv
- Contents v
- About the Authors xxx
- About the Technical Reviewers xxxi
- Acknowledgments xxxii
- Preface xxxiii
- Chapter 1: Getting Started With Entity Framework 1
- Chapter 2: Entity Data Modeling Fundamentals 9
- Chapter 3: Querying an Entity Data Model 63
- Chapter 4: Using Entity Framework in ASP.NET 115
- Chapter 5: Loading Entities and Navigation Properties 155
- Chapter 6: Beyond the Basics with Modeling and Inheritance 189
- Chapter 7: Working with Object Services 251
- Chapter 8: Plain Old CLR Objects 271
- Chapter 9: Using the Entity Framework in N-Tier Applications 311
- Chapter 10: Stored Procedures 359
- Chapter 11: Functions 393
- Chapter 12: Customizing Entity Framework Objects 429
- Chapter 13: Improving Performance 471
- Chapter 14: Concurrency 509
- Chapter 15: Advanced Modeling 529
- Index 591

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Contents

- **Contents at a Glance** iv
- **Contents** v
- **About the Authors** xxx
- **About the Technical Reviewers** xxxi
- **Acknowledgments** xxxii
- **Preface** xxxiii

Who This Book Is For xxxiii
What's in This Book xxxiv
About the Recipes xxxv
Stuff You Need to Get Started xxxvi
Code Examples xxxvi
The Database xxxvi
Apress Website xxxvii

- **Chapter 1: Getting Started With Entity Framework** 1

A Brief Tour of the Entity Framework World 2
Models 2
Terminology 3
Code 4
Visual Studio 2010 4
Using Entity Framework 5

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

■ **Chapter 2: Entity Data Modeling Fundamentals 9**

2-1. Creating a Simple Model 9

Problem 9

Solution 9

How It Works 14

2-2. Creating a Model from an Existing Database 16

Problem 16

Solution 16

How It Works 19

2-3. Modeling a Many-to-Many Relationship with No Payload 22

Problem 22

Solution 22

How It Works 23

2-4. Modeling a Many-to-Many Relationship with a Payload 26

Problem 26

Solution 26

How It Works 27

2-5. Modeling a Self-Referencing Relationship 29

Problem 29

Solution 29

How It Works 30

2-6. Splitting an Entity Across Multiple Tables 33

Problem 33

Solution 33

How It Works 35

2-7. Splitting a Table Across Multiple Entities 37

Problem 37

Solution 37

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 40
2-8. Modeling Table per Type Inheritance 42
Problem 42
Solution 42
How It Works 44
2-9. Using Conditions to Filter an ObjectSet 46
Problem 46
Solution 46
How It Works 48
2-10. Modeling Table per Hierarchy Inheritance 49
Problem 49
Solution 49
How It Works 52
2-11. Modeling Is-a and Has-a Relationships Between Two Entities 54
Problem 54
Solution 55
How It Works 56
2-12. Creating, Modifying, and Mapping Complex Types 57
Problem 57
Solution 58
How It Works 59
■ Chapter 3: Querying an Entity Data Model 63
3-1. Executing an SQL Statement 63
Problem 63
Solution 63
How It Works 65
3-2. Returning Objects from a SQL Statement 66
Problem 66

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution 66
How It Works 67
3-3. Returning Objects from an Entity SQL Statement 68
Problem 68
Solution 68
How It Works 70
3-4. Specifying Fully Qualified Names in Entity SQL 71
Problem 71
Solution 72
How It Works 74
3-5. Finding a Master that Has Detail in a Master-Detail Relationship 74
Problem 74
Solution 74
How It Works 76
3-6. Setting Default Values in a Query 77
Problem 77
Solution 77
How It Works 79
3-7. Returning Multiple Result Sets From a Stored Procedure 80
Problem 80
Solution 80
How It Works 81
3-8. Comparing Against a List of Values 82
Problem 82
Solution 82
How It Works 84
3-9. Building and Executing a Query Against an ObjectSet<T> 85
Problem 85

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution 85
How It Works 87
3-10. Returning a Primitive Type From a Query 87
Problem 87
Solution 87
How It Works 89
3-11. Filtering Related Entities 89
Problem 89
Solution 89
How It Works 92
3-12. Applying a Left Outer Join 93
Problem 93
Solution 93
How It Works 95
3-13. Ordering by Derived Types 96
Problem 96
Solution 96
How It Works 97
3-14. Paging and Filtering 98
Problem 98
Solution 98
How It Works 100
3-15. Grouping by Date 101
Problem 101
Solution 101
How It Works 102
3-16. Flattening Query Results 103
Problem 103

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution 103
How It Works 105
3-17. Grouping by Multiple Properties 105
Problem 105
Solution 106
How It Works 108
3-18. Using Bitwise Operators in a Filter 108
Problem 108
Solution 108
How It Works 111
3-19. Joining on Multiple Columns 111
Problem 111
Solution 111
How It Works 113
■ Chapter 4: Using Entity Framework in ASP.NET 115
4-1. Building a Search Query 115
Problem 115
Solution 115
How It Works 118
4.2. Building CRUD Operations in an ASP.NET Web Page 119
Problem 119
Solution 119
How It Works 124
4-3. Executing Business Logic When Changes Are Saved 124
Problem 124
Solution 124
How It Works 126
4-4. Loading Related Entities 127

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Problem 127
Solution 127
How It Works 129
4-5. Searching with QueryExtender 129
Problem 129
Solution 129
How It Works 135
4-6. Retrieving a Derived Type Using an EntityDataSource Control 136
Problem 136
Solution 136
How It Works 139
4-7. Filtering with ASP.NET's URL Routing 139
Problem 139
Solution 139
How It Works 142
4-8. Building CRUD Operations with an ObjectDataSource Control 143
Problem 143
Solution 143
How It Works 148
4-9. Using Entity Framework With MVC 149
Problem 149
Solution 149
How It Works 154
■ **Chapter 5: Loading Entities and Navigation Properties 155**
5-1. Loading Related Entities 155
Problem 155
Solution 155
How It Works 158

Licensed by Matt Allen 2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5-2. Loading a Complete Object Graph 160
Problem 160
Solution 160
How It Works 162
5-3. Loading Navigation Properties on Derived Types 162
Problem 162
Solution 163
How It Works 164
5-4. Using Include() with Other LINQ Query Operators 165
Problem 165
Solution 165
How It Works 166
5-5. Deferred Loading of Related Entities 167
Problem 167
Solution 167
How It Works 169
5-6. Filtering and Ordering Related Entities 169
Problem 169
Solution 170
How It Works 171
5-7. Executing Aggregate Operations on Related Entities 172
Problem 172
Solution 172
How It Works 174
5-8. Testing Whether an Entity Reference or Entity Collection Is Loaded 174
Problem 174
Solution 174
How It Works 176

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5-9. Loading Related Entities Explicitly 176
Problem 176
Solution 176
How It Works 178
5-10. Filtering an Eagerly Loaded Entity Collection 180
Problem 180
Solution 180
How It Works 181
5-11. Using Relationship Span 182
Problem 182
Solution 182
How It Works 184
5-12. Modifying Foreign Key Associations 184
Problem 184
Solution 184
How It Works 187
■ Chapter 6: Beyond the Basics with Modeling and Inheritance 189
6-1. Retrieving the Link Table in a Many-to-Many Association 189
Problem 189
Solution 189
How It Works 191
6-2. Exposing a Link Table as an Entity 192
Problem 192
Solution 192
How It Works 195
6-3. Modeling a Many-to-Many, Self-Referencing Relationship 196
Problem 196
Solution 196

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 197
6-4. Modeling a Self-Referencing Relationship Using Table per Hierarchy Inheritance 200
Problem 200
Solution 200
How It Works 202
6-5. Modeling a Self-Referencing Relationship and Retrieving a Complete Hierarchy . 204
Problem 204
Solution 204
How It Works 207
6-6. Mapping Null Conditions in Derived Entities 208
Problem 208
Solution 208
How It Works 209
6-7. Modeling Table per Type Inheritance Using a Non-Primary Key Column 211
Problem 211
Solution 211
How It Works 215
6-8. Modeling Nested Table per Hierarchy Inheritance 216
Problem 216
Solution 216
How It Works 218
6-9. Limiting the Values Assigned to a Foreign Key 220
Problem 220
Solution 220
How It Works 222
6-10. Applying Conditions in Table per Type Inheritance 224
Problem 224
Solution 224

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 225
6-11. Creating a Filter on Multiple Criteria 227
Problem 227
Solution 227
How It Works 229
6-12. Using Complex Conditions with Table per Hierarchy Inheritance 232
Problem 232
Solution 233
How It Works 235
6-13. Modeling Table per Concrete Type Inheritance 238
Problem 238
Solution 238
How It Works 240
6-14. Applying Conditions on a Base Entity 242
Problem 242
Solution 242
How It Works 244
6-15. Creating Independent and Foreign Key Associations 246
Problem 246
Solution 246
How It Works 247
6-16. Changing an Independent Association into a Foreign Key Association 247
Problem 247
Solution 248
How It Works 249
■ Chapter 7: Working with Object Services 251
7-1. Dynamically Building a Connection String 251
Problem 251

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution 251
How It Works 252
7-2. Reading a Model from a Database 253
Problem 253
Solution 253
How It Works 256
7-3. Deploying a Model 257
Problem 257
Solution 257
How It Works 257
7-4. Using the Pluralization Service 258
Problem 258
Solution 258
How It Works 260
7-5. Retrieving Entities from the Object State Manager 261
Problem 261
Solution 261
How It Works 263
7-6. Generating a Model from the Command Line 263
Problem 263
Solution 263
How It Works 264
7-7. Working with Dependent Entities in an Identifying Relationship 264
Problem 264
Solution 264
How It Works 267
7-8. Inserting Entities Using an Object Context 267
Problem 267

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution 267
How It Works 269
■ **Chapter 8: Plain Old CLR Objects 271**
8-1. Using POCO 271
Problem 271
Solution 271
How It Works 276
8-2. Loading Related Entities With POCO 276
Problem 276
Solution 276
How It Works 279
8-3. Lazy Loading With POCO 279
Problem 279
Solution 280
How It Works 282
8-4. POCO With Complex Type Properties 283
Problem 283
Solution 283
How It Works 285
8-5. Notifying Entity Framework About Object Changes 286
Problem 286
Solution 286
How It Works 288
8-6. Retrieving the Original (POCO) Object 289
Problem 289
Solution 289
How It Works 291
8-7. Manually Synchronizing the Object Graph and the Object State Manager 292

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Problem 292
Solution 292
How It Works 295
8-8. Testing Domain Objects 296
Problem 296
Solution 296
How It Works 304
8-9. Testing a Repository Against a Database 305
Problem 305
Solution 305
How It Works 308
■ **Chapter 9: Using the Entity Framework in N-Tier Applications 311**
9-1. Deleting an Entity When Disconnected 311
Problem 311
Solution 311
How It Works 314
9-2. Managing Concurrency When Disconnected 315
Problem 315
Solution 315
How It Works 318
9-3. Finding Out What Has Changed 319
Problem 319
Solution 319
How It Works 322
9-4. Using POCO With WCF 323
Problem 323
Solution 323
How It Works 328

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

9-5. Using Self-Tracking Entities With WCF 329
Problem 329
Solution 329
How It Works 333
9-6. Validating Self-Tracking Entities 334
Problem 334
Solution 334
How It Works 338
9-7. Using Self-Tracking Entities on the Server Side 338
Problem 338
Solution 338
How It Works 344
9-8. Serializing Proxies in a WCF Service 345
Problem 345
Solution 345
How It Works 349
9-9. Serializing Self-Tracking Entities in the ViewState 349
Problem 349
Solution 349
How It Works 353
9-10. Fixing Duplicate References on a WCF Client 354
Problem 354
Solution 354
How It Works 357
■ Chapter 10: Stored Procedures 359
10-1. Returning an Entity Collection 359
Problem 359
Solution 359

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 361
10-2. Returning Output Parameters 362
Problem 362
Solution 362
How It Works 365
10-3. Returning a Scalar Value Result Set 365
Problem 365
Solution 365
How It Works 367
10-4. Returning a Complex Type from a Stored Procedure 367
Problem 367
Solution 367
How It Works 369
10-5. Defining a Custom Function in the Storage Model 370
Problem 370
Solution 370
How It Works 372
10-6. Populating Entities in a Table per Type Inheritance Model 373
Problem 373
Solution 373
How It Works 375
10-7. Populating Entities in a Table per Hierarchy Inheritance Model 376
Problem 376
Solution 376
How It Works 378
10-8. Mapping the Insert, Update, and Delete Actions to Stored Procedures 379
Problem 379
Solution 379

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 381

10-9. Using Stored Procedures for the Insert and Delete Actions in a Many-to-Many Association 382

Problem 382

Solution 383

How It Works 387

10-10. Mapping the Insert, Update, and Delete Actions to Stored Procedures for Table per Hierarchy Inheritance 387

Problems 387

Solution 387

How It Works 391

■ **Chapter 11: Functions 393**

11-1. Returning a Scalar Value from a Model Defined Function 393

Problem 393

Solution 393

How It Works 396

11-2. Filtering an Entity Collection Using a Model Defined Function 397

Problem 397

Solution 397

How It Works 400

11-3. Returning a Computed Column from a Model Defined Function 401

Problem 401

Solution 401

How It Works 404

11-4. Calling a Model Defined Function from a Model Defined Function 404

Problem 404

Solution 404

How It Works 408

11-5. Returning an Anonymous Type From a Model Defined Function 408

Licensed by Matt Allen 2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Problem 408
Solution 408
How It Works 411
11-6. Returning a Complex Type From a Model Defined Function 412
Problem 412
Solution 412
How It Works 415
11-7. Returning a Collection of Entity References From a Model Defined Function 415
Problem 415
Solution 415
How It Works 417
11-8. Using Canonical Functions in eSQL 418
Problem 418
Solution 418
How It Works 419
11-9. Using Canonical Functions in LINQ 419
Problem 419
Solution 419
How It Works 421
11-10. Calling Database Functions in eSQL 422
Problem 422
Solution 422
How It Works 424
11-11. Calling Database Functions in LINQ 424
Problem 424
Solution 424
How It Works 425
11-12. Defining Built-in Functions 425

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Problem 425
Solution 426
How It Works 428

■ **Chapter 12: Customizing Entity Framework Objects 429**

12-1. Executing Code When SaveChanges() Is Called 429
Problem 429
Solution 429
How It Works 431

12-2. Validating Property Changes 432
Problem 432
Solution 432
How It Works 434

12-3. Logging Database Connections 435
Problem 435
Solution 435
How It Works 437

12-4. Recalculating a Property Value When an Entity Collection Changes 437
Problem 437
Solution 437
How It Works 439

12-5. Automatically Deleting Related Entities 440
Problem 440
Solution 440
How It Works 443

12-6. Deleting All Related Entities 443
Problem 443
Solution 444
How It Works 447

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

12-7. Assigning Default Values 447
Problem 447
Solution 447
How It Works 450
12-8. Retrieving the Original Value of a Property 451
Problem 451
Solution 451
How It Works 453
12-9. Retrieving the Original Association for Independent Associations 454
Problem 454
Solution 454
How It Works 457
12-10. Retrieving XML 457
Problem 457
Solution 457
How It Works 460
12-11. Applying Server-Generated Values to Properties 460
Problem 460
Solution 460
How It Works 464
12-12. Validating Entities on SavingChanges 464
Problem 464
Solution 464
How It Works 469
■ Chapter 13: Improving Performance 471
13-1. Optimizing Queries in a Table per Type Inheritance Model 471
Problem 471
Solution 471

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 472
13-2. Retrieving a Single Entity Using an Entity Key 473
Problem 473
Solution 473
How It Works 474
13-3. Retrieving Entities for Read Only 475
Problem 475
Solution 475
How It Works 476
13-4. Improving the Startup Time 477
Problem 477
Solution 477
How It Works 478
13-5. Efficiently Building a Search Query 479
Problem 479
Solution 480
How It Works 481
13-6. Making Change Tracking with POCO Faster 482
Problem 482
Solution 482
How It Works 485
13-7. Compiling LINQ Queries 485
Problem 485
Solution 485
How It Works 488
13-8. Returning Partially Filled Entities 489
Problem 489
Solution 489

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works 491
13-9. Moving an Expensive Property to Another Entity 491
Problem 491
Solution 491
How It Works 494
13-10. Avoiding Include 495
Problem 495
Solution 495
How It Works 497
13-11. Improving QueryView Performance 497
Problem 497
Solution 497
How It Works 499
13-12. Generating Proxies Explicitly 500
Problem 500
Solution 500
How It Works 502
13-13. Preventing the Update of All Columns in Self-Tracking Entities 503
Problem 503
Solution 503
How It Works 507
■ **Chapter 14: Concurrency 509**
14-1. Applying Optimistic Concurrency 509
Problem 509
Solution 509
How It Works 511
14-2. Managing Concurrency When Using Stored Procedures 512
Problem 512

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution 512
How It Works 516
14-3. Reading Uncommitted Data 516
Problem 516
Solution 516
How It Works 518
14-4. Implementing the "Last Record Wins" Strategy 518
Problem 518
Solution 518
How It Works 520
14-5. Getting Affected Rows from a Stored Procedure 520
Problem 520
Solution 521
How It Works 524
14-6. Optimistic Concurrency with Table Per Type Inheritance 524
Problem 524
Solution 524
How It Works 527
14-7. Generating a Timestamp Column with Model First 527
Problem 527
Solution 527
How It Works 528
■ Chapter 15: Advanced Modeling 529
15-1. Creating an Association on a Derived Entity 529
Problem 529
Solution 529
How It Works 531
15-2. Mapping an Entity to Customized Parts of One or More Tables 532

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Problem 532
Solution 532
How It Works 534
15-3. Creating Conditional Associations 536
Problem 536
Solution 536
How It Works 541
15-4. Fabricating Additional Inheritance Hierarchies 542
Problem 542
Solution 542
How It Works 545
15-5. Sharing Audit Fields Across Multiple Entities 547
Problem 547
Solution 547
How It Works 551
15-6. Modeling a Many-to-Many Relationship with Payload 553
Problem 553
Solution 553
How It Works 555
15-7. Mapping a Foreign Key Column to Multiple Associations 556
Problem 556
Solution 557
How It Works 562
15-8. Using Inheritance to Map a Foreign Key Column to Multiple Associations 564
Problem 564
Solution 564
How It Works 567
15-9. Creating Read-only and Computed Properties 568

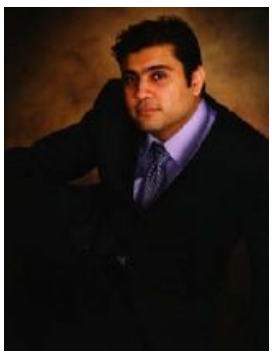
Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Problem 568
Solution 568
How It Works 574
15-10. Mapping an Entity to Multiple Tables 576
Problem 576
Solution 576
How It Works 577
15-11. Mapping an Entity to Multiple Entity Sets (MEST) 578
Problem 578
Solution 578
How It Works 583
15-12. Extending Table per Type with Table per Hierarchy 585
Problem 585
Solution 585
How It Works 588
■ **Index 591**

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

About the Authors

■ **Larry Tenny** has more than 20 years of experience developing applications using a broad range of development tools primarily targeting the Windows platform. He has extensive .NET development experience from its initial community preview as Next Generation Windows Services to the latest .NET 4.0 release. He has a PhD in Computer Science from Indiana University.



■ **Zeeshan Hirani** is a longtime .NET and database developer. He is a senior developer at a top Internet e-commerce site using Entity Framework, ASP.NET, Silverlight, and many other Microsoft technologies. He has extensive experience with many ORM and database technologies, which provides him with a unique perspective on Microsoft's Entity Framework. He has written several articles, maintains an influential Entity Framework blog, and is a frequent contributor to many .NET forums. He is a Microsoft MVP.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

About the Technical Reviewers

Licensed by Matt Allen

David Annesley-DeWinter has worked with and implemented business solutions on the .NET platform since .NET 1.1, leveraging a background in rich data modeling using ORM (object-role modeling) to drive data requirements for organizations. After working in industry on a variety of applications ranging from highly concurrent middleware services to customer-facing rich client and web applications, David moved to Washington and now works on the Entity Framework team at Microsoft, where he focuses on features for object services like the POCO templates and Code First. In his spare time, David enjoys photography and rowing for the Sammamish Rowing Association in Redmond. You can read more about his experiences with the Entity Framework and other .NET-related topics on his blog at `http://blogs.rev-net.com/ddewinter/` and follow him on Twitter at `@ddewinter`.

Brian Swan spent 14 years teaching high-school and junior-college mathematics and dabbled in teaching introductory computer science courses before making the jump to a career in technology. After a brief stint at Amazon Web Services as a support engineer, he joined Microsoft where he has been focused on learning and writing about various data access technologies. In his spare time he is an amateur husband, father, mountain biker, back packer, runner, and beer drinker.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Acknowledgments

Without a doubt, this book required an enormous amount of research and countless hours of discovering the common problems that developers face with Entity Framework. For months, we pestered the Entity Framework Development Team and others in the forums and in hundreds of e-mails. We found the team and the Internet community extremely patient and willing to help us explore this technology and understand the problems developers often encounter. For this we are deeply grateful.

In particular, we would like to thank Diego Vega, Microsoft Program Manager, for sharing his valuable knowledge and expertise. Throughout the writing of this book, Diego shed light into many areas where our knowledge fell short. Without his explanations we would not have been able to deliver the breadth and depth of content represented in this book.

We also would like to thank Noam Ben-Ami, Microsoft Program Manager, who answered many of our questions about Entity Framework Designer. Noam provided incredibly important insight into some of the most interesting aspects of Entity Framework.

We would like to thank our technical editors, David Annesley-DeWinter and Brian Swan, for their careful and meticulous review of every recipe. The technical reviewers worked through each recipe and provided us with very valuable advice throughout the process.

So much of what this book is is due to the professionalism and guidance of the many people at Apress. Our editor, Jonathan Gennick, is not only a helpful guide but also a good friend and a writing mentor. Our coordinating editor, Mary Tobin, provided the steady guidance and clock-like cadence that kept this massive project humming along even when the writers struggled to keep up. Nancy Sixsmith, our very patient copy editor, checked (and often corrected) every word with machine-like precision. To all the wonderful people at Apress, thank you so much.

Finally, we want to thank our families, friends, and co-workers for putting up with a couple of overly excited developers turned writers.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Preface

Anyone who has been developing on the Microsoft platform for the last several years knows the drill: every few years, there's a new database access technology. There was ODBC; then DAO and RDO; OLEDB, ADO, and ADO.NET; LINQ to SQL; and now Entity Framework! In many ways, this progression of technologies has been confusing, but in other ways it's wonderfully refreshing to see this field evolve from simple open connectivity to componentized connectivity, to disconnected access in a managed environment, to friction-less access syntax, and finally to conceptual modeling.

It's the conceptual modeling that is the defining feature of Entity Framework and is at the heart of this book. Entity Framework builds upon the previous data access paradigms providing an environment that supports rich, real-world domain level modeling. We can now think of and program against real-world things such as orders and customers, and leverage concepts such as inheritance to reason about things in our domain and not just rows and columns.

There is no question that Entity Framework is the future of data access for the Microsoft platform. The first release in August of 2008 was widely considered a good first step. Now, more than year later, this new release of Entity Framework (often called EF 4.0) as part of the newly released Visual Studio 2010 and .NET 4.0 has matured into a full function data access technology ready for production use in both green field and legacy applications.

The concepts and patterns you will learn as you use the recipes in this book will serve you well into the future as Microsoft continues to evolve Entity Framework in the years to come.

Who This Book Is For

This book is for anyone who develops applications for the Microsoft platform. All of us who work in this field need access to data in our applications. We are all interested in more powerful and intuitive ways to reason about and program against the real-world objects in our applications. It makes much more sense for us to architect, design, and build applications in terms of customers, orders, and products rather than rows and columns scattered among tables locked away in a database. Because we can reason about problem space in terms of real-world objects, we have a lot more confidence in our design and in the code that we build. We are also better able to document and explain our applications to others. This makes our code much more maintainable.

Entity Framework is not just for developers. Microsoft is aggressively positioning the modeling concepts in Entity Framework to serve as the conceptual domain for Reporting Services and Integration Services as well as other technologies that process, report on, and transform data. Entity Framework is quickly becoming a core data access foundation for many other Microsoft technologies.

This book contains well over 150 recipes that you can put to work right away. Entity Framework is a large and complex topic. Perhaps it's too big for a monolithic reference book. In this book, you will find direct and self-contained answers to just about any problem you're facing in building your Entity Framework-powered applications. Along the way, you'll learn an enormous amount about Entity Framework.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

What's in This Book

We've organized the recipes in this book by topic. Sometimes we've found that a recipe fits into more than one chapter, and sometimes we find that a recipe doesn't fit perfectly in any chapter. We think it's better to include all the important recipes rather than just the ones that fit, so you might find yourself wondering why a particular recipe is in a certain chapter. Don't worry. If you find the recipe useful, we hope that you can forgive its (mis)placement. At least we got it into the book.

The following is a list of the chapters and a brief synopsis of the recipes you'll find in them:

Chapter 1: Getting Started with Entity Framework. We explain the motivation behind Entity Framework. We also explain what the framework is and what it does for you.

Chapter 2: Entity Data Modeling Fundamentals. This chapter covers the basics in modeling. Here you'll find out how to get started with modeling and with Entity Framework in general. If you're just getting started, this chapter probably has the recipes you're looking for.

Chapter 3: Querying an Entity Data Model. We'll show you how to query your model using both LINQ to Entities and Entity SQL.

Chapter 4: Using Entity Framework in ASP.NET. Web applications are an important part of the development landscape, and Entity Framework is ideally suited for ASP.NET. In this chapter we focus on using the EntityDataSource to interact with your model for selects, inserts, updates, and deletes.

Chapter 5: Loading Entities and Navigation Properties. The recipes in this chapter cover just about every possibility for loading entities from the database.

Chapter 6: Beyond the Basics with Modeling and Inheritance. Modeling is a key part of Entity Framework. This is the second of three chapters with recipes specifically about modeling. In this chapter, we included recipes that cover many of the more complicated, yet all-too-common modeling problems you'll find in real-world applications.

Chapter 7: Working With Object Services. In this chapter, we included recipes that provide practical solutions for the deployment of your models. We also provide recipes for using the Pluralization Service, using the edmgen.exe utility, and working with so-called *identifying relationships*.

Chapter 8: Plain Old CLR Objects. Using code-generated entities is fine in many scenarios, but there comes a time when you need to use your own classes as EntityTypes. The recipes in this chapter cover plain old CLR objects (POCO) in depth. They show you how to use your own classes and reduce code dependence on Entity Framework.

Chapter 9: Using Entity Framework in n-Tier Applications. The recipes in this chapter cover a wide range of topics using Entity Framework across the wire. We cover POCO, self-tracking entities, serialization, and concurrency.

Chapter 10: Stored Procedures. If you are developing or maintaining a real-world, data-centric application, you most likely work with stored procedures. The recipes in this chapter show you how to consume the data exposed by those stored procedures.

Chapter 11: Functions. The recipes in this chapter show you how to create and use model-defined functions. We also show how to use functions provided by Entity Framework, as well as functions exposed by the storage layer.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Chapter 12: Customizing Entity Framework Objects. The recipes in this chapter show you how to respond to key events, such as when objects are persisted. We also show how to customize the way those events are handled.

Chapter 13: Improving Performance. For many applications, getting the best performance possible is an important goal. This chapter shows you several ways to improve the performance of your Entity Framework applications.

Chapter 14: Concurrency. Lots of instances of your application are changing the database. How do you control who wins? The recipes in this chapter show you how to manage concurrency.

Chapter 15: Advanced Modeling. This is the last of three chapters that focuses on modeling. The recipes in this chapter show you how to solve some of the most vexing modeling problems you are ever likely to encounter.

About the Recipes

At present there are four perspectives on model development in Entity Framework. Each of these perspectives is at a different level of maturity in the product and at a different level of use in the community.

The initial perspective supported by Entity Framework is called Database First. Using Database First, a developer starts with an existing database that is used to create an initial conceptual model. This initial model serves as the starting point for further development. As changes occur in the database, the model can be updated from these database changes. Database First was the initial perspective supported in Entity Framework, is the best-supported approach, and is widely used to migrate existing applications to Entity Framework.

The current release of Entity Framework introduced the Model First perspective. With Model First, the developer starts with a blank design surface and creates a conceptual model. Once the conceptual model is complete, Entity Framework can automatically generate a script to create a complete database for the conceptual model. In this release there is limited support for many of the modeling scenarios. As you might expect, realizing an arbitrarily complex conceptual model in a traditional relational database is an enormous challenge. The support for this perspective will mature over time and will likely become the dominant approach, particularly for new projects.

Persistence ignorance, which is supported in many ORM products, is now supported in the current version of Entity Framework. With persistence ignorance, you can use plain old CLR objects, usually referred to as POCO, as entity types. There is no need for them to inherit from `EntityObject`. We have devoted a number of recipes to POCO.

Finally, an emerging perspective is Code First. In this approach, there is no .edmx file (which encapsulates model and mapping information). Your objects create and use a model dynamically at runtime. This perspective is still in the experimental stage and is available as a Community Technology Preview.

In this book, we focus on the Database First perspective. This perspective is the most widely used and most mature approach. Many, if not most, developers in the Entity Framework community find themselves working with existing applications or developing models that are not readily supported by the other perspectives. We also have to share a dirty little secret: many existing applications don't exactly use the best database designs. Way too often we find ourselves working with databases (of course, created by other less talented developers) that are poorly designed. As developers, sometimes in larger organizations with lots of process control, or with lots of fragile legacy code, we can't change the database enough to really fix the design. In these cases, we simply have to work with the database design we have.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Many of the recipes we selected for this book take on the task of modeling some of these more challenged database designs. We've found hundreds of examples of these databases in the wild and we've worked with many developers in the Entity Framework community who have struggled to model these databases. We've taken these experiences and selected a number of recipes that will help you solve these problems.

Stuff You Need to Get Started

Okay, what do you need? First off, you will need Microsoft's latest software development environment. Microsoft's Visual Studio 2010 comes complete with full support for Entity Framework. Visual Studio 2010 Express Edition is freely available. The other versions of Visual Studio fully support Entity Framework.

You'll need a database. Microsoft SQL Server 2008 with Service Pack 1 is the simplest choice, but there are Entity Framework providers for databases from other vendors. Microsoft SQL Server 2008 Express is freely available. Make sure you apply the latest service packs and updates. These recipes were built and tested using Microsoft SQL Server 2008. Previous versions of SQL Server or other databases may not play well with a few of the recipes.

Code Examples

This book is all about recipes that solve very specific problems in a way that allows you to directly apply the solution to your code. Feel free to use and adapt any of the code you find here to help build or maintain your applications. Of course, it's not okay to copy large parts of this material and distribute it for fun or profit. If you need to copy large parts of this material, contact our publisher, Apress, to get permission.

If you use our code publicly (in blogs, forums, and so on), we would appreciate, but don't require, some modest attribution such as author, title, and ISBN.

We've taken a decidedly low-tech approach in the code in each recipe. We've tried not to clutter the code with unnecessary constructs and clever tricks. In the text, we show just the code of interest, but we also show enough to give the proper context. In the download for the code, we have complete solutions for each recipe. The solutions build simple applications that you can modify and run over and over to play with various changes that suit your needs.

The Database

Of course, there is more to each recipe than just the code. We created a single database for all the recipes. This makes it much easier to work through the recipes because there is just one database to create in your development environment.

To keep some sanity in the table names and provide at least a little organization, we created a schema for each chapter. The recipes in the chapter use the tables in the corresponding schema. In the text, we often show database diagrams similar to the one in Figure 0-1. This helps make clear the table structure we're working with. Each table in a diagram is annotated (courtesy of SQL Server Management Studio) with the name of the table and the schema for the table. Because we reuse table names throughout the book (we're just not creative enough not to), this helps keep straight exactly which tables we're referring to in the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

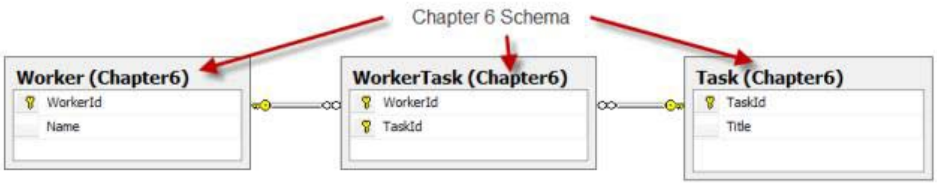


Figure 0-1. Each database diagram in the text has the schema name next to the table name.

We've also provided the complete set of database diagrams for each recipe as part of the database. If something isn't clear from just the tables, especially when several tables are involved, it often helps to look at the diagram to sort things out.

Apress Website

Visit the Apress website (`http://apress.com/book/view/1430227036`) for the complete code download as well as the database with all the tables and database diagrams used in this book. Please look for the "Source Code" link underneath the cover image.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227036
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 1

Getting Started With Entity Framework

In relational databases, we think of things in terms of tables with rows and columns. Tables are very structured and amenable to all sorts of interesting set theory. Before the dawn of object-oriented programming, back in the day when we focused on "structured" programming and wrote function after function, it seemed a good idea to break down a big problem into lots of little problems. Working with tables, rows, and columns seemed a good match with our code. Our code was structured and procedural. Our data was structured and backed up by database side procedures. Things lined up well. Many database vendors even supplied preprocessors that allowed developers to intermix SQL statements and C (or Fortran) code. Life was good for a time.

Much has evolved on the code side. Now we think in terms of objects in a domain model. We architect, design, and program against real-world things like customers and orders. We draw the nouns in our problem space on whiteboards. We draw lines between them, denoting relationships and interactions between customers and orders. We build specifications and assign work to development teams in terms of these drawings. In short, we architect, design, and program at a conceptual level that is very distant from the logical organization of the database.

While the software development process has matured, and the way in which we reason about and solve problems during the process has evolved, the data in our databases has been locked in the same tables, rows, and columns structure. The synergy between structured data and our code evaporated as quickly as structured programming in the heat of modern object-oriented development. To cope with this growing mismatch, many projects introduced a "database layer" to isolate the object-oriented code from the data store. This layer translated objects to the rows and columns saved in tables. Many commercial solutions were introduced, including an entire field of Object Relational Mapping (ORM). These tools provided many out-of-the-box yet configurable ways to bridge the ever-widening gap between the evolving development process and structured data.

The fundamental problem is this: the gap is widening and it is increasingly impractical to fill the gap with yet more glue code. No one wants to develop and maintain more in house glue code and commercial solutions are struggling to keep up. Microsoft is fond of calling this gap the *impedance mismatch* between code and data.

Microsoft's Entity Framework, together with Language-Integrated Query (LINQ) and a new query language called Entity SQL, are technologies specifically designed to address the impedance mismatch problem. With Entity Framework, we model the nouns (entity types) on a design surface. On the design surface, we can model the relationships (associations) between entities. In our code, we program against these entities and associations. LINQ allows us to express the set theoretical concepts of relational databases directly in our code while working in terms of entity types and associations. All this elevates the interactions with the data store to the conceptual level we design and reason about with our code. When we work at the conceptual level for both code and data, we can worry less about the logical schema of the data store and free ourselves from the noise of the glue code and third-party ORM tools.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

A Brief Tour of the Entity Framework World

Entity Framework is a collection of technologies for developing applications that use data. Unlike previous Microsoft data-access technologies, Entity Framework, together with Visual Studio, is a comprehensive, model-based ecosystem that you can use to develop a wide range of data-oriented applications. You can develop desktop applications, server-side applications, Internet applications using ASP.NET and Silverlight, and Windows Communication Foundation (WCF)-based multimachine applications. In this book, we have recipes that will help you develop all these types of applications.

Let's take a very brief look at some of the parts of the Entity Framework ecosystem. What follows is not by any means a comprehensive description of Entity Framework; that would take hundreds of pages. We'll look at just a few key things to help get you oriented for the recipes that are at the heart of this book.

Models

Entity Framework is a technology that's all about modeling. The modeling in Entity Framework represents more of an evolutionary point than a revolutionary idea. As you work with models in Entity Framework, you will see many familiar genetic markers from previous technologies and patterns. You will, no doubt, see a family resemblance to entity-relationship diagrams and the long-used conceptual, logical, and physical design layers approach.

Entity Framework uses models characterized by the Entity Data Model (EDM). The EDM is a formal structure for defining data used in the applications you create with Entity Framework. The EDM defines the data types, the specific definitions of what types of relationships are allowed, the schemas that support the model, and the mapping between these schemas. The models you build and program against are defined in terms of the EDM, but are not themselves Entity Data Models. It's sort of like the difference between a class and an instance of the class (an object). If you understand the definition of a class, you know a great deal about the behavior of an instance of the class, but the two are quite different. Although models in applications are often confused with the EDM (sometimes even in Microsoft's documentation), it's important to emphasize the difference.

So what does the EDM say about the structure of models? Well, quite a lot as it turns out. First off, a model is composed of three layers: a conceptual layer, a storage layer, and a mapping layer. The syntax for each layer, that is, how it's represented in a file, is XML-based. The schema for each of these layers is defined, of course, by the EDM. XML is amenable to designer applications, can be consumed and produced by developer tools, and is sort of human-readable. For convenience, all three of these layers are usually bundled in single file in your project. The file has the .edmx extension.

The *conceptual layer*, or *conceptual model*, is perhaps the only part many developers see when they work with a model. Visual Studio provides a full-featured designer that enables you to whiteboard the high-level nouns (entity types) and relationships (associations) in our domain. On the conceptual layer, there is no taint of a physical storage organization. With the designer you create entities, perhaps establish inheritance hierarchies, and link entities together through associations. The syntax for the conceptual model is defined by the Conceptual Schema Definition Language (CSDL).

Every useful application needs to persist objects to some data store. The *store layer*, or *store model*, defines the data store. This includes the tables, columns, and data types that the EntityClient layer will ultimately map to the underlying database. The syntax for the store model is defined in by the Store Schema Definition Language (SSDL).

The mapping between the conceptual model and the store model is defined by the *mapping layer*. Among other things, this layer defines how properties on entities map to columns on tables. Although it is tucked away in the Mapping Details window, this mapping layer is also exposed to the developer

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

through the designer. The syntax for the mapping layer is defined by the Mapping Specification Language (MSL).

Terminology

There is a huge amount of terminology around Entity Framework. If you have used any of the popular ORM tools or are familiar with database modeling, you've probably encountered some of the terminology before. Much of the terminology is unique to Entity Framework. Here we'll provide just a few of the basic terms to get us started.

An *EntityType* is the "noun" in your model. An EntityType, like a class, defines a new type. An instance of an EntityType is referred to as an *entity*. An EntityType is represented on the design surface as a box with various properties. Figure 1-1 shows two EntityTypes: Employee and Task.

Licensed by Matt Allen 2175280

Figure 1-1. *A model with Employee and Task and a one-to-many association between them*

Most of the time, in this book and throughout much of the documentation, blogs, forum posts, and so on, you'll find people referring to a particular EntityType as just an entity. Following this somewhat lazy approach, people will say that Figure 1-1 has two entities: Employee and Task, when really it has two EntityTypes. For this section, we'll stick with *EntityType* to be clear.

An EntityType usually has one or more *properties*. Just like with a class, a property is a named value with a specific data type. Properties can have simple types like integer, string, and so on; or have ComplexTypes; or be collections. *Navigation* properties refer to other entities in an association. The non-navigation properties on an EntityType are usually just called *scalar* properties.

An *Association* between two EntityTypes is shown on the design surface as a line connecting the EntityTypes. The line is annotated to show the multiplicity on each end of the association. The association in Figure 1-1 is a one-to-many association between Employee and Task. An Employee can have zero or more tasks. Each Task is associated to exactly one Employee.

Every EntityType has some set of properties which denote its *EntityKey*. An EntityKey for an entity uniquely indentifies the entity to Entity Framework and is most often obtained from the entity's representation in the underlying database.

An *EntitySet* holds instances of an EntityType (or one of its derived types) at runtime. In most cases, instances of a given EntityType are held in just one EntitySet, but we'll cover examples of Multiple EntitySets per Type (MEST).

A *ComplexType* is a set of related properties. A ComplexType does not have a key like an EntityType does. A ComplexType is typically used to group related properties together to be reused in a model or to simplify a model. For example, an Address ComplexType might group together address line 1, address line 2, suite number, city, state, and ZIPCode. In a Customer EntityType, you might have a BillingAddress and a ShipToAddress, each of type Address. This makes Customer a little simpler because it doesn't need individual properties for each part of the customer's address (see Figure 1-2).

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 1-2. *Address ComplexType that is the data type for both the BillingAddress and the ShipToAddress for the customer*

Code

The story of Entity Framework is not quite complete without code. After all, Entity Framework is just a tool you use to create your applications. The models, EntityTypes, associations, mappings, and so on are ultimately expressed in concrete code that becomes part of your application. This code is either generated by Visual Studio and Entity Framework during the build process or else it is created by you, the developer. You get to choose quite a bit about the code-generation process or the lack of it by changing various properties on your project and modifying or creating code-generation templates.

Visual Studio uses a relatively new code-generation technology called *Text Template Transformation Toolkit*, which is such mouthful everyone refers to it simply as *T4 Templates*. T4 Template support comes from the Domain Specific Language tools work that Microsoft has been doing for some time. It is a way to provide a human-readable template that used to guide the tooling in Visual Studio when it automatically generates code. The great thing about T4 Template support in Visual Studio is that you can edit the templates to tailor the code-generation process to your needs. This is an advanced technique, but it is necessary in some cases. We'll show you how to do this in a few recipes.

Creating your own classes that implement your EntityTypes is often referred to as using Plain Old CLR Objects or simply POCO. Your POCO classes typically don't have any dependence on Entity Framework plumbing. The recipes in Chapter 8 show you the basics of creating and using POCO. There are also many recipes throughout the book that show you how to use POCO in specific contexts such as in n-tier applications.

The workflow for code generation is managed during the build process by Windows Workflow Foundation (WF). Although WF has been around for a while, the .NET 4.0 release contains a completely re-engineered implementation and now an integration into Visual Studio 2010.

Visual Studio 2010

Of course, the main tool we use when developing applications for the Windows environment is Visual Studio. This Integrated Development Environment has evolved over many years from a simple C++ compiler and editor to a highly integrated, multilanguage environment that supports the entire software development lifecycle. Visual Studio and its related tools and services provide for design, development, unit testing, debugging, software configuration management, build management and continuous integration, and much more. Don't be worried if you haven't used all these in your work; few developers have. The point is that Visual Studio 2010 is a full-featured toolset. Visual Studio plays a vital role in the development of Entity Framework applications.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Visual Studio provides an integrated design surface for Entity Framework models. Using this design surface and other tools in Visual Studio, you can create models from scratch or create them from an existing database.

If you have an existing database, which is the case for many of us with existing applications, Visual Studio provides tools for importing your tables and relationships into a model. This approach, known as *Database First*, is the best-supported modeling approach and the one we use in most of the recipes in this book. This fits nicely with the real world because few of us have the luxury of developing brand-new applications. Most of us have to extend, maintain, or evolve our existing code and databases.

When you create a model from scratch, also known as *Model First*, you start with an empty design surface and add new EntityTypes to the surface and create both the associations and inheritance hierarchies for your model. When you are done creating the model, right-click the design surface and select Generate Database from Model. Not all modeling scenarios are currently supported with Model First. In time, Model First will likely become the dominant modeling approach. For now, Model First is great for many modeling scenarios for new applications.

Once you have created your model, changes often happen. That's the nature of software development. Visual Studio provides tools for updating the model from the database. This will keep the model synchronized with changes in the database. Beware: updating the model will update the conceptual model (.csdl), but may also change the underlying store model (.ssdl). This is fine if you haven't manually edited the store model (which we do in some recipes). If you have edited the store model, you may find that you have to reapply your changes. We'll warn you in the recipes in this book when updating the model will make changes to your store model.

Using Entity Framework

Entity Framework is an integral part of Visual Studio 2010. One simple way to start using Entity Framework is to include a new ADO.NET Entity Data Model in your project. Right-click your project and select Add ➤New Item. In the dialog box (see Figure 1-3), choose the ADO.NET Entity Data Model template. This template is located under the Data templates. Click Add to launch the Entity Data Model Wizard.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

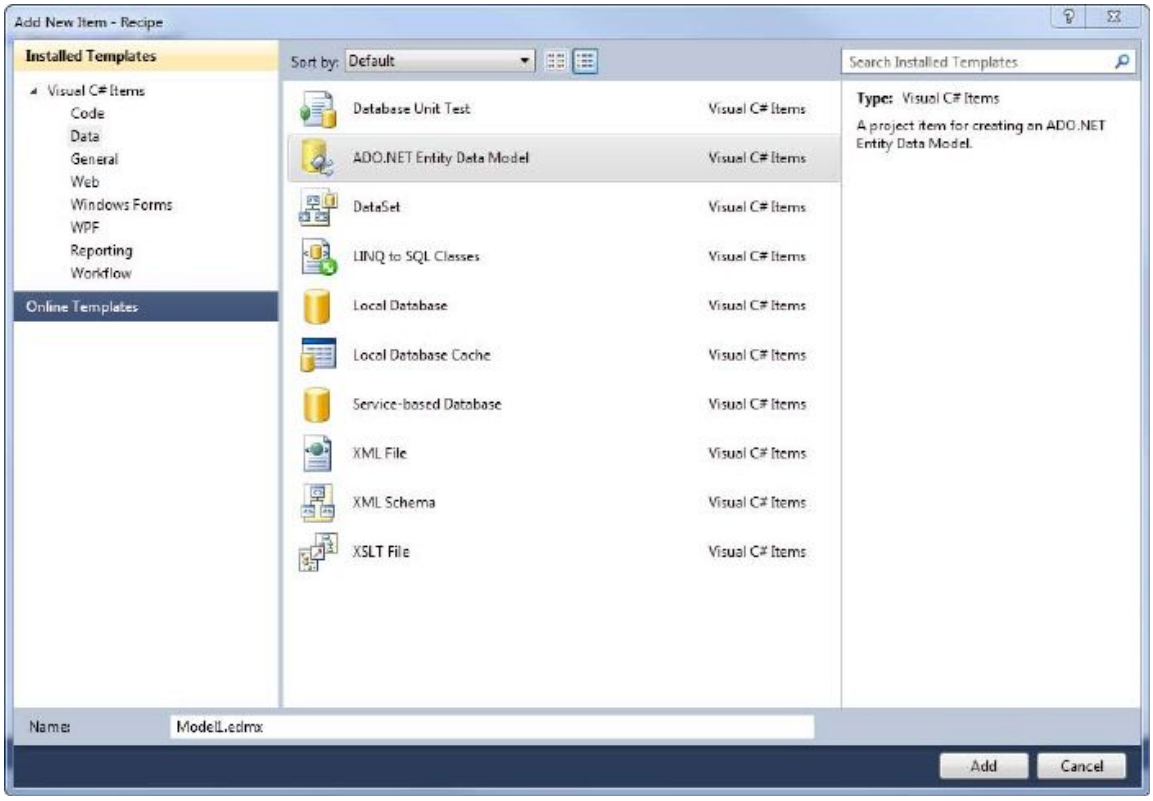


Figure 1-3. *Adding a new model to your project*

There are two options on the first page of the Entity Data Model Wizard: start with an existing database or start with an empty model. This first page is shown in Figure 1-4.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

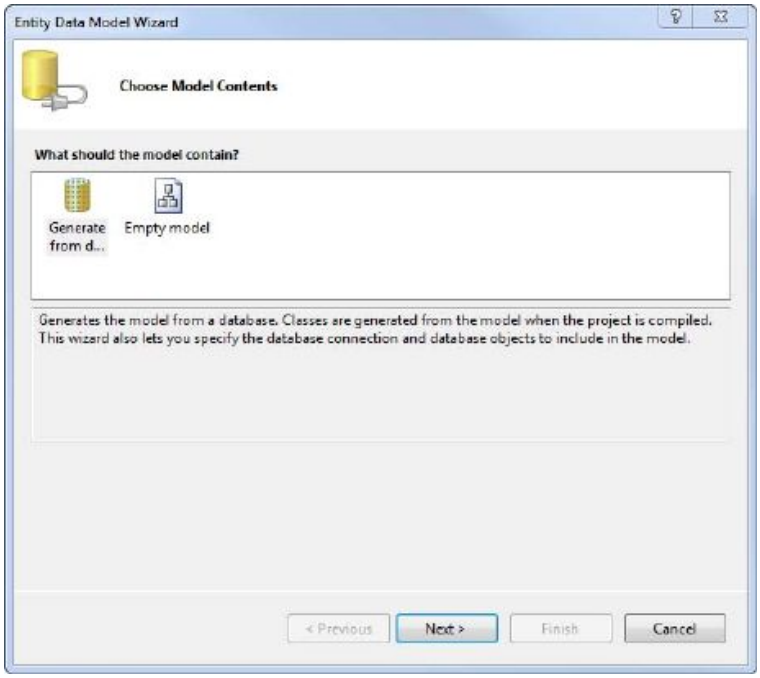


Figure 1-4. *The Entity Data Model Wizard gives you a choice between creating a model from an existing database or starting with an empty model*

Generating a model from an existing database is the Database First approach. The wizard will create a model based on one or more tables, views, and stored procedures from an existing database. With just a few exceptions, the tables you include will be modeled as EntityTypes. If the tables you include are related in the database, these relationships will be modeled as associations. This is a great way to create your model if you already have a database for your application. If you're working on a brand-new application, you may want to start with an empty model. This approach is called Model First.

With Model First, you are presented with an empty design surface. Right-click the design surface to create new EntityTypes, associations, or inheritances. You can also drag them from the Toolbox onto the design surface. Once your model is complete, just right-click the design surface and select Generate Database from Model. This will generate a script you can use to create the database tables and relationships for the model.

With either Model First or Database First, you use the designer to develop your model. The key parts of a model in the designer are shown in Figure 1-5. In this model, a Customer is in a one-to-many association with an Order. Each customer may have many orders, but each order is associated with just one customer. The Mapping Details window shows that the Customer EntityType maps to the Customer table in the database. The Mapping Detail window also shows the mapping between the columns in the Customer table and the scalar properties in the Customer EntityType.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

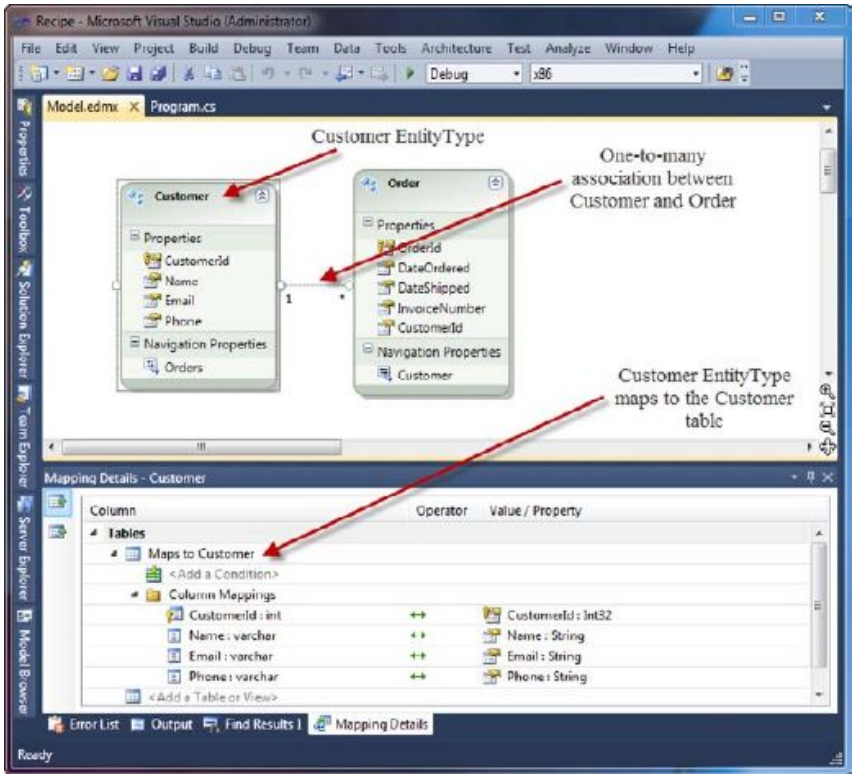


Figure 1-5. *Key parts of a model in the designer*

Of course, there is a lot more to the designer and the model than just the few key parts illustrated in Figure 1-5. In the recipes in this book, we'll cover just about every aspect of using the designer to create models. In some cases, we go beyond what can be done with the designer and show you how to create models that require directly editing the underlying .edmx file. The .edmx file contains the complete model definition, including the conceptual layer, store layer, and mapping layer.

Okay, now that you have a model, how do you program against it? Well, that's where Entity Framework really shines: you program against objects in the model (EntityTypes) as you do with other objects in your application. For the model in Figure 1-5, your code uses Customer and Order in much the same way as you use other objects.

If you want to insert a new customer and order into the database, you can use the `new` operator to create instances of the Customer and Order types, set the properties, add them to the in-memory context that represents the model, and call `SaveChanges()`. All the necessary SQL code is generated and sent to the database to insert the rows. To retrieve customers and orders from the database, you use either LINQ or Entity SQL to create a query in terms of the EntityTypes and associations in the model.

The recipes throughout this book will show you step by step how to model just about every conceivable database scenario; how to query, insert, update, and delete using these models; and how to use Entity Framework in many kinds of applications.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 2

Entity Data Modeling Fundamentals

Entity Framework is a new technology from Microsoft. More likely than not, you are just beginning to explore Entity Framework and you are probably asking the question, "Okay, how do I get started?" If this describes you, this chapter is a great place to start. If, on the other hand, you have built some working models and feel comfortable with a few key modeling concepts such as entity splitting and inheritance, you can skip this chapter.

In this chapter, we will walk you through the basic examples of modeling with Entity Framework. Modeling is the core feature of Entity Framework and what distinguishes Entity Framework from previous Microsoft data access platforms. Once you have built your model, you can write code against the model rather than against the rows and columns in the relational database.

We start off this chapter with an example of how to create a simple conceptual model and let Entity Framework create the underlying database. In the remaining examples, we will show you how to create models from existing tables and relationships in your databases.

2-1. Creating a Simple Model

Problem

You have a brand new project and want to create a model with just one entity.

Solution

Let's imagine you want to create an application to hold names and phone numbers of people you know. To keep things simple, let's assume you need just one entity type: Person.

To create the new model, do the following:

1. Right-click your project and select Add ➤ New Item.
2. From the templates, select ADO.NET Entity Data Model and click Add. This template is located in Data under Visual C# Items. See Figure 2-1.
3. In the first step of the wizard, choose Empty Model and click Finish. The wizard will create a new conceptual model with an empty design surface.
4. Right-click the design surface and select Add ➤ Entity.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5. Type Person in the Entity name field and select the box to Create a key property. Use Id as the Key Property. Make sure its Property Type is `Int32`. Click OK, and a new Person entity will appear on the design surface. See Figure 2-2.
6. Right-click near the top of the Person entity and select Add ➤ Scalar Property. A new scalar property will be added to the Person entity.
7. Rename the scalar property **FirstName**. Add scalar properties for LastName, MiddleName, and PhoneNumber.
8. Right-click the Id property and select Properties. In the properties view, change the StoreGeneratedPattern property to Identity. This flags the Id property as a value that will be computed by the store layer (database). The database script we get at the end will flag the Id column as an identity column, and the storage model will know that the database will automatically manage the values in this column.

The completed conceptual model should look like the model in Figure 2-3.

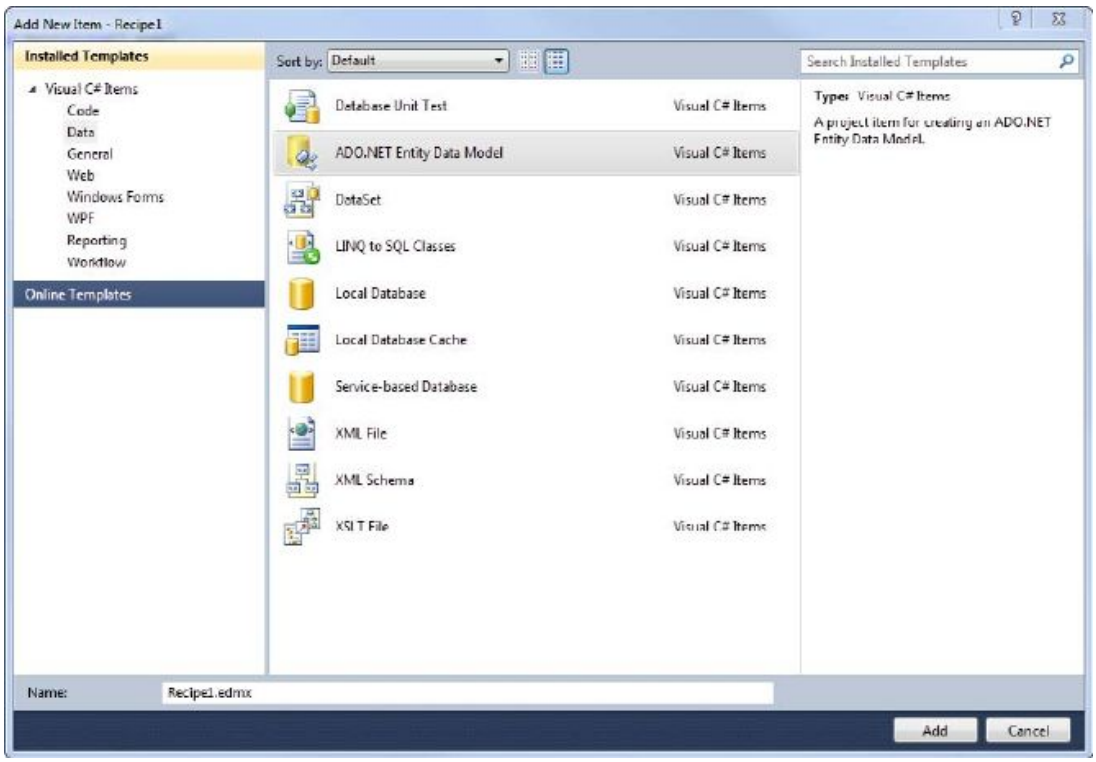


Figure 2-1. *Adding a new .emdx file that contains XML describing the conceptual model, storage model, and mapping layer*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

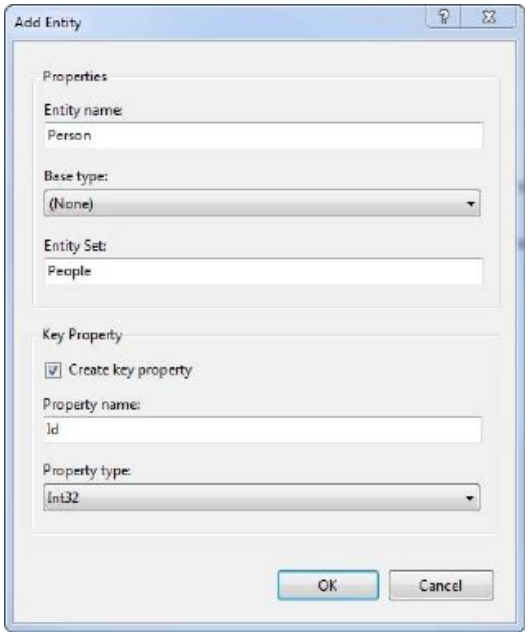


Figure 2-2. *Adding a new entity type representing a Person in our conceptual model*

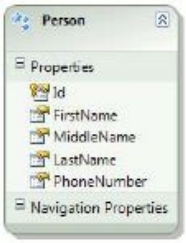


Figure 2-3. *Our completed model with an entity type representing a Person*

You now have a simple conceptual model. To generate a database for our model, there are a few things we still have to do:

9. We need to change a couple of properties of our model to help with the housekeeping. Right-click the design surface and select properties. Change the Database Schema Name to **Chapter2** and change the Entity Container Name to **EFRecipesEntities**. Figure 2-4 illustrates these changes.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10. Right-click the design surface and select Generate Database Script from Model. Select an existing database connection or create a new one. In Figure 2-5, we've opted to create a new connection to our local machine and to the database EFRecipes.
11. Click OK to complete the connection properties and click Next to preview the database script (see Figure 2-6). Once you click Finish, the generated script is added to your project.
12. Run the database script in a query window to create the People table.

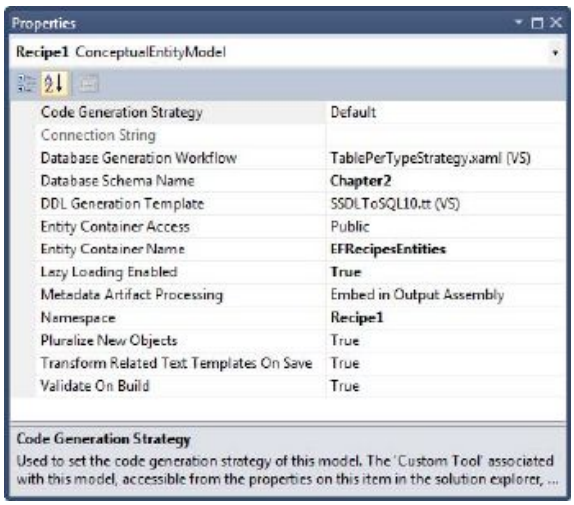
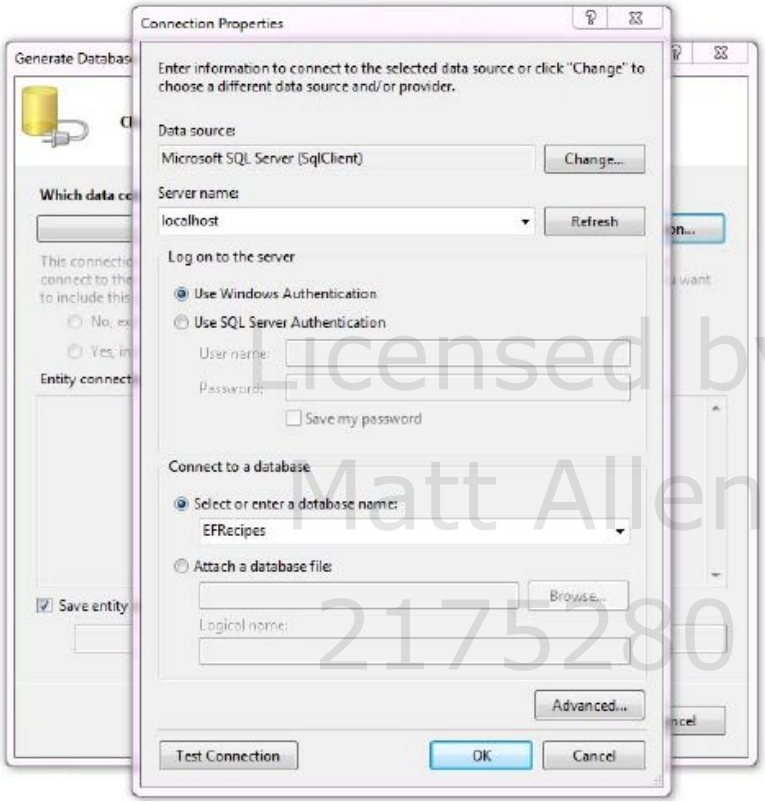


Figure 2-4. *Changing the properties of our model*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



Licensed by Matt Allen 2175280

Figure 2-5. *Creating a new database connection that will be used by Entity Framework to create a database script that we can use to create a database from our conceptual model*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

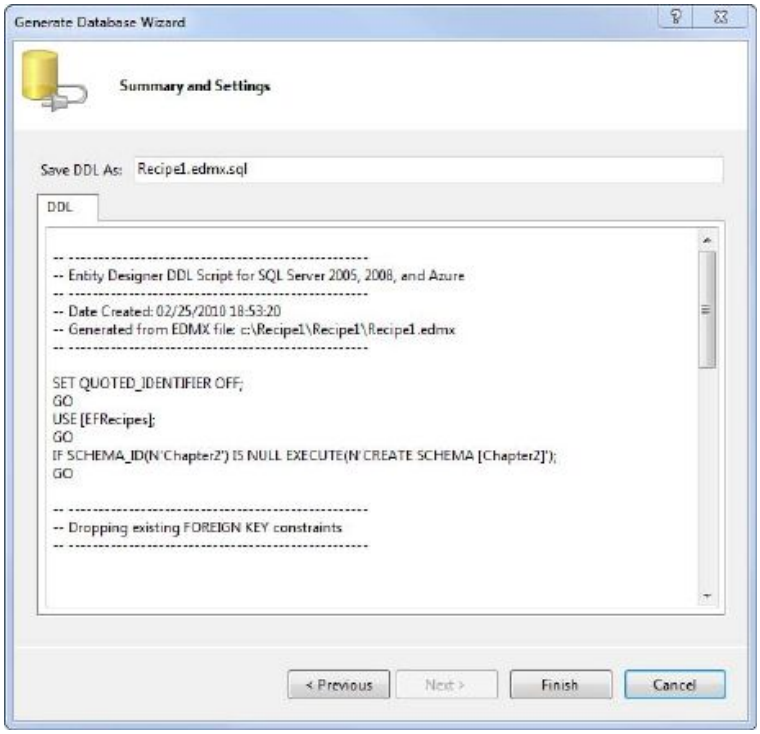


Figure 2-6. *Generating the storage model in the .edmx file and creating the database script*

How It Works

The Entity Framework Designer is a powerful tool for creating and updating a conceptual model, storage model, and mapping layer. This tool provides support for bidirectional model development. You can either start with a clean design surface and create a model; or start with a database you already have and import it to create a conceptual model, storage model, and mapping layer. The current version of the Designer supports somewhat limited roundtrip modeling, allowing you to re-create your database from a model and update the model from changes in your database.

The model has a number of properties that affect what goes in the generated storage model and database script. We changed two of these properties. The first was the name of the container. This is the class derived from `ObjectContext`. We called this EFRecipesEntities to be consistent with the contexts we use throughout this book.

Additionally, we changed the schema to "Chapter 2." This represents the schema used to generate the storage model as well as the database script.

The code in Listing 2-1 demonstrates one simple way to create and insert instances of our Person entity type. The code also demonstrates iterating through all the Person entities in our database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 2-1. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var person = new Person() { FirstName = "Robert", MiddleName="Allen",
                                LastName = "Doe", PhoneNumber = "867-5309" };
    context.People.AddObject(person);
    person = new Person() { FirstName = "John", MiddleName="K.",
                            LastName = "Smith", PhoneNumber = "824-3031" };
    context.People.AddObject(person);
    person = new Person() { FirstName = "Billy", MiddleName="Albert",
                            LastName = "Minor", PhoneNumber = "907-2212" };
    context.People.AddObject(person);
    person = new Person() { FirstName = "Kathy", MiddleName="Anne",
                            LastName = "Ryan", PhoneNumber = "722-0038" };
    context.People.AddObject(person);

    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var person in context.People)
    {
        System.Console.WriteLine("{0} {1} {2}, Phone: {3}",
                                 person.FirstName, person.MiddleName,
                                 person.LastName, person.PhoneNumber);
    }
}
```

The output of the code in Listing 2-1 should look something like the following:

```
John K. Smith, Phone: 824-3031

Robert Allen Doe, Phone: 867-5309

Kathy Anne Ryan, Phone: 722-0038

Billy Albert Minor, Phone: 907-2212
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Best Practice

When we created a new instance of the object context, we did it within a `using()` statement:

```
using (var context = new EFRecipesEntities())
{
…
}
```

If you are not familiar with this pattern, it's really pretty simple. Normally, when we get a new instance of an object, we use the new operator and assign the result to some variable. When the variable goes out of scope and the object is not longer referenced by anything else, the garbage collector will do its job at some point and reclaim the memory for the object. That works great for most of the objects we create in our .NET applications because most objects hold on to resources that can wait around for whenever the garbage collector has a chance to reclaim them. The garbage collector is rather nondeterministic. It reclaims resources pretty much on its own schedule, which we can only partially influence.

Instances of `ObjectContext` hold on to resources such as database connections that we want to release as soon as we're done with them. We don't really want these database connections to stay open waiting for the garbage collector to eventually reclaim them.

There are a few nice features of `using` statements. First, when the code execution leaves the `using() {}` block, the `Dispose()` method on the context will be called (because `ObjectContext` implements the IDisposable interface). For `ObjectContext`, the `Dispose()` method closes any active database connections and properly cleans up any other resources that need to be released.

Second, no matter how the code leaves the `using(){}` block, the `Dispose()` method is called. Most importantly, this includes return statements and exceptions that may be thrown within the code block. The `using(){}` block is kind of a guarantee that critical resources will be reclaimed properly.

The best practice here is to always wrap your code in the `using(){}` block when creating new instances of `ObjectContext`. It's one more step to help bullet-proof your code.

2-2. Creating a Model from an Existing Database

Problem

You have an existing database with a few tables, perhaps a few views, and some foreign key constraints, and you want to create a model for this database.

Solution

Let's say you have database describing poets and their poetry. Your relational database might look something like the diagram in Figure 2-7.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

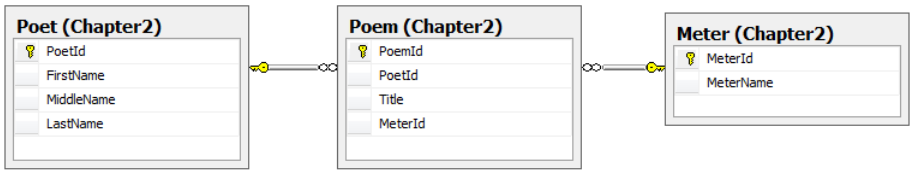


Figure 2-7. *A simple database for poets and their poetry*

From this database diagram, you can see that a poet can be the author of one or more poems and each poem can be categorized by its meter, which is the basic pattern of a poem's verse. It's not shown in this diagram, but our database also has a view that joins the tables together so that we can more easily enumerate each poet, poem, as well as the poem's meter.

To import the view, tables, and relationships into a model, do the following:

1. Right-click your project and select Add ➤ New Item.
2. From the Visual C# Items Data templates, select ADO.NET Entity Data Model.
3. Select Generate from database to create the model from our existing tables. Click Next.
4. Either choose an existing connection to your database or create a new connection. If you are creating a new connection, you will need to select your database server, your authentication method (Windows or SQL Server), and the database. Once you have selected these, it's a good idea to click Test Connection to be sure the connection is ready to go. Once you have tested the connection, click Next.
5. The next dialog box shows all the tables, views, and stored procedures in the database. Check the items you want to include in the model. We want to select all the tables (Meter, Poem, and Poet). We also want to select the view (vwLibrary). For now, leave the two check boxes for pluralizing and including foreign key columns selected. We'll talk more about them in a minute. Figure 2-8 shows the things we've selected.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

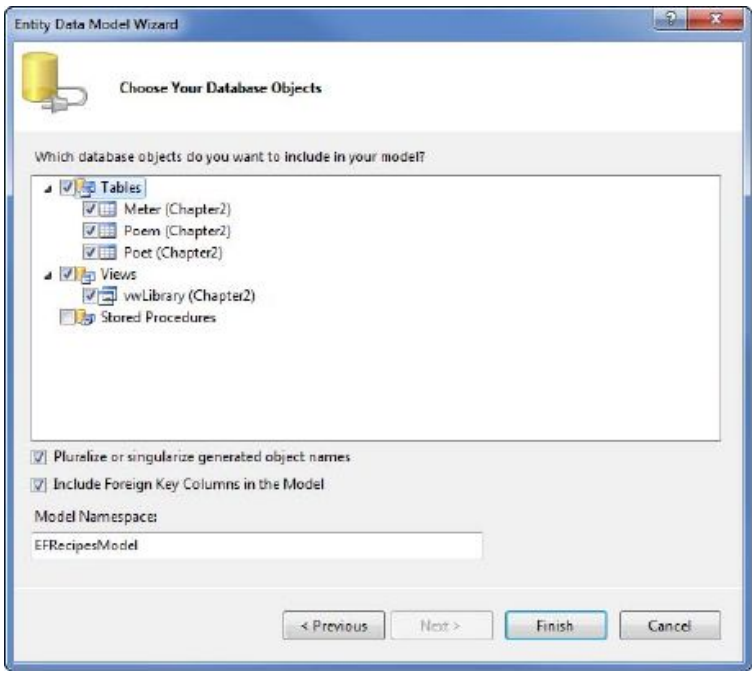


Figure 2-8. *Selecting the tables and view to include in our model. Leave the Pluralize or singularize generated object names and Include Foreign Key Columns in the Model checked.*

When you click Finish, the wizard will create a new model with our three tables and the view. The wizard will also read the foreign key constraints from the database and infer a one-to-many relationship between Poet and Poem(s) as well as a one-to-many relationship between Meter and Poem(s).

Figure 2-9 shows the new model created for us by including the Poet, Poem, and Meter tables as well as the vwLibrary view.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

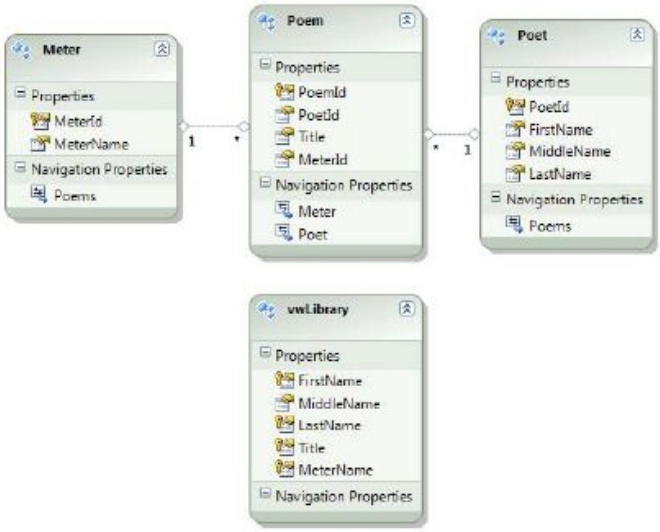


Figure 2-9. *Our completed model*

You now have a model you can use in your code. Note that the vwLibrary entity is based on the vwLibrary view in our database. In most databases, views are read only objects: inserts, deletes, and updates are typically not supported at the database layer. This is also the case with Entity Framework. Entity Framework considers views read only. You can get around this by mapping stored procedures for the create, update, and delete actions for view-based entities. We will show you how to do just that in Chapter 6.

How It Works

Let's look at the model created for us by the importing process. First, notice that the entities have scalar properties and navigation properties. The scalar properties map to the columns in the tables of the database while the navigation properties are derived from the relationships between the tables.

In our database diagram, a poem has a meter and a poet (the author). These correspond to the Meter and Poet navigation properties. If we have an instance of a Poem entity, the Poet navigation property holds an instance of a Poet entity while the Meter navigation property holds an instance of a Meter entity. A poet can be the author of any number of poems. The Poems navigation property contains a collection of instances of the Poem entity. This collection can be empty, of course, for those poets that have yet to write any poetry. For the Meter entity, the Poems navigation property is also a collection. For this navigation property, the collection holds instances of Poems that have the given meter. Our database did not contain any relationships with the vwLibrary view and our model reflects this with an empty set of navigation properties on the vwLibrary entity.

Notice that the Import Wizard was smart enough to pluralize the navigation properties that contained collections. If you right-click the entities and look at their properties, you will notice that the entity set names for each of the entities is also property pluralized. For example, the entity set name for

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

the Poem entity is Poems. This automatic pluralization happened because we left the Pluralize or singularize generated object names option checked.

The Include Foreign Key Columns in the model option caused the foreign keys to be included in the model as well. Although it may seem a little unnecessary to have both foreign keys and navigation properties, we'll see in many of the following recipes that having direct access to the foreign keys can be useful.

The code in Listing 2-2 demonstrates how to create instances of Poet, Poem, and Meter entities in our model and how to save these entities to our database. The code also shows you how to query the model to retrieve the poets and poems from the database.

In the first block of code in Listing 2-2, we create instances of the Poet, Poem, and Meter entity types for the poet John Milton, his poem "Paradise Lost," and the meter for the poem—which in this case is Iambic Pentameter. Once we have created the instances of the entity types, we set the poem's Meter property to the meter instance and the poem's Poet property to the poet instance. Using the same approach we build up the other entities relating each poem to its meter and poet. Once we have everything in place, we call **SaveChanges()** to generate and execute the appropriate SQL statements to insert the rows into the underlying database.

Listing 2-2. Inserting into and querying our model

```
using (var context = new EFRecipesEntities())
{
    var poet = new Poet { FirstName = "John", LastName = "Milton" };
    var poem = new Poem { Title = "Paradise Lost" };
    var meter = new Meter { MeterName = "Iambic Pentameter" };
    poem.Meter = meter;
    poem.Poet = poet;
    context.Poems.AddObject(poem);
    poem = new Poem { Title = "Paradise Regained" };
    poem.Meter = meter;
    poem.Poet = poet;
    context.Poems.AddObject(poem);

    poet = new Poet { FirstName = "Lewis", LastName = "Carroll" };
    poem = new Poem { Title = "The Hunting of the Shark" };
    meter = new Meter { MeterName = "Anapestic Tetrameter" };
    poem.Meter = meter;
    poem.Poet = poet;
    context.Poems.AddObject(poem);

    poet = new Poet { FirstName = "Lord", LastName = "Byron" };
    poem = new Poem { Title = "Don Juan" };
    poem.Meter = meter;
    poem.Poet = poet;
    context.Poems.AddObject(poem);

    context.SaveChanges();

}

using (var context = new EFRecipesEntities())
{
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    var poets = from p in context.Poets select p;
    foreach (var poet in poets)
    {
        Console.WriteLine("{0} {1}", poet.FirstName, poet.LastName);
        foreach (var poem in poet.Poems)
        {
            Console.WriteLine("\t{0} ({1})", poem.Title, poem.Meter.MeterName);
        }
    }
}

// using our vwLibrary view
using (var context = new EFRecipesEntities())
{
    var items = from i in context.vwLibraries select i;
    foreach (var item in items)
    {
        Console.WriteLine("{0} {1}", item.FirstName, item.LastName);
        Console.WriteLine("\t{0} ({1})", item.Title, item.MeterName);
    }
}
```

The output from the code in Listing 2-2 is the following:

```
Lord Byron

        Don Juan (Anapestic Tetrameter)

Lewis Carroll

        The Hunting of the Shark (Anapestic Tetrameter)

John Milton

        Paradise Regained (Iambic Pentameter)

        Paradise Lost (Iambic Pentameter)

Lewis Carroll

        The Hunting of the Shark (Anapestic Tetrameter)

Lord Byron

        Don Juan (Anapestic Tetrameter)

John Milton
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        Paradise Regained (Iambic Pentameter)

John Milton

        Paradise Lost (Iambic Pentameter)
```

In the code, we start by creating and initializing instances of the poet, poem, and meter for the first of John Milton's poems. Once we have these in place, we set the poem's Meter navigation property and the poem's Poet navigation property to the instances of poem and meter. Now that we have the poem instance completed, we add it using the **AddToPoems()** method. Entity Framework does all the remaining work of adding the poem to the Poems collection on the poet instance and adding the poem to the Poems collection on the meter instance. The rest of the setup follows the same pattern. To shorten the code, we reuse variables and instances where we can.

Once we have the all the objects created and all the navigation properties initialized, we have completed the object graph. Entity Framework keeps track of the changes we've made to build the object graph. These changes are tracked in the object context. Our **context** variable contains an instance of the object context (it's of type **ObjectContext**) and is what we used to build the object graph. To send these changes to the database, we call the **SaveChanges()** method.

To query our model and, of course, verify that we did indeed save everything to the database, we grab a fresh instance of the object context and query it using LINQ to Entities. We could have reused the same instance of the object context, but then we know it has the object graph and any subsequent queries we do against it won't flow through to the database because the graph is already in memory.

Using LINQ to Entities, we query for all the poets, and for each poet we print out the poet's name and the details for each of their poems. The code is pretty simple, but it does use a couple of nested **for** loops.

The last block of code uses the vwLibrary entity. This entity is based on our vwLibrary view. This view joins the tables together to flatten things out a bit and provide a cleaner perspective. When we query for each poet against the vwLibraries entity set, we can get by with just one **for** loop. The output is a little different because we repeat the poet's name for each poem.

There is one last thing to note in this example. We didn't insert the poets, poems, and meters using the vwLibrary entity because views are always read-only in most database systems. In Entity Framework, we can't insert (or update, or delete) entities that are based on views. Of course, we'll show you exactly how to overcome this little challenge in many of the recipes in this book!

2-3. Modeling a Many-to-Many Relationship with No Payload

Problem

You have a couple of tables in an existing database that are related to each other via a link or junction table. The link table contains just the foreign keys used to link the two tables together into a many-to-many relationship. You want to import these tables model this many-to-many relationship.

Solution

Let's say your database tables look something like the database diagram in Figure 2-10.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

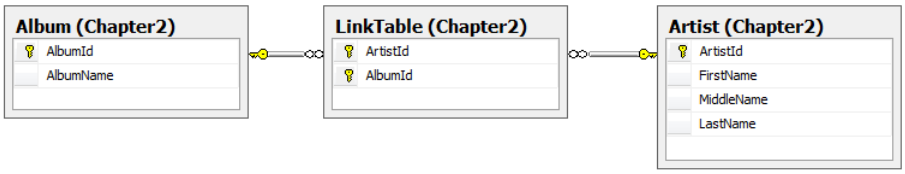


Figure 2-10. *Artists and albums in a many-to-many relationship*

To create a model and import these tables and relationships, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Items Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the tables Album, LinkTable, and Artist. Leave the Pluralize and Foreign Key options checked. Click Finish.

Licensed by Matt Allen 21/5280

The wizard will create the model shown in Figure 2-11.

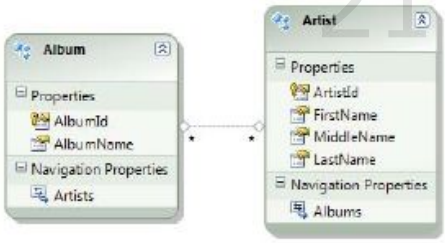


Figure 2-11. *Our model with a many-to-many relationship between our tables*

The many-to-many relationship between Album and Artist is represented by a line with the * character on both ends. Because an Album can have many Artists and an Artist can responsible for many Albums, each of these navigation properties is of type EntityCollection.

How It Works

In Figure 2-11, an artist can be related to many albums, whereas an album can be the work of many artists. Notice that the link table from Figure 2-10 is not represented as an entity in our model. Because our link table has no scalar properties (that is, it has no payload), Entity Framework assumes that its sole purpose is to create the association between Album and Artist. If the link table had scalar properties, Entity Framework would have created a very different model, as we will see in the next Recipe.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The code in Listing 2-3 demonstrates how to insert new albums and artists into our model and how to query our model both for artists and their albums and albums with their artists.

Listing 2-3. Inserting and querying our artists and albums model through the many-to-many association

```
using (var context = new EFRecipesEntities())
{
    // add an artist with two albums
    var artist = new Artist { FirstName = "Alan", LastName = "Jackson" };
    var album1 = new Album { AlbumName = "Drive" };
    var album2 = new Album { AlbumName = "Live at Texas Stadium" };
    artist.Albums.Add(album1);
    artist.Albums.Add(album2);
    context.Artists.AddObject(artist);

    // add an album for two artists
    var artist1 = new Artist { FirstName = "Tobby", LastName = "Keith" };
    var artist2 = new Artist { FirstName = "Merle", LastName = "Haggard" };
    var album = new Album { AlbumName = "Honkytonk University" };
    artist1.Albums.Add(album);
    artist2.Albums.Add(album);
    context.Albums.AddObject(album);

    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Artists and their albums...");
    var artists = from a in context.Artists select a;
    foreach (var artist in artists)
    {
        Console.WriteLine("{0} {1}", artist.FirstName, artist.LastName);
        foreach (var album in artist.Albums)
        {
            Console.WriteLine("\t{0}", album.AlbumName);
        }
    }

    Console.WriteLine("\nAlbums and their artists...");
    var albums = from a in context.Albums select a;
    foreach (var album in albums)
    {
        Console.WriteLine("{0}", album.AlbumName);
        foreach (var artist in album.Artists)
        {
            Console.WriteLine("\t{0} {1}", artist.FirstName, artist.LastName);
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The output from the code in Listing 2-3 looks like the following:

```
Artists and their albums...

Alan Jackson

        Drive

        Live at Texas Stadium

Tobby Keith

        Honkytonk University

Merle Haggard

        Honkytonk University


Albums and their artists...

Drive

        Alan Jackson

Live at Texas Stadium

        Alan Jackson

Honkytonk University

        Tobby Keith

        Merle Haggard
```

After getting an instance of our object context, we create and initialize an instance of an Artist entity type and a couple of instances of the Album entity type. We add the albums to the artist and then add the artist to the Object Context.

Next, we create and initialize a couple instances of the Artist entity type and an instance of the Album entity type. Because the two artists collaborated on the album, we add the album to both artists' Albums navigation property (which is of type EntityCollection). Adding the album to the Object Context causes the artists to get added as well.

Now that the completed object graph is part of the object context, the only thing left to do is to use **SaveChanges()** to save the whole thing to the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When we query the database in a brand new Object Context, we grab the artists and display their albums. Then we grab the albums and print the artists that created the albums.

Notice that we never refer to the underlying LinkTable from Figure 2-10. In fact, this table is not even represented in our model as an entity. The LinkTable is represented in the many-to-many association which we access via the Artists and Albums navigation properties.

2-4. Modeling a Many-to-Many Relationship with a Payload

Problem

You have a many-to-many relationship in which the link table contains some payload data (any additional columns beyond the foreign keys) and you want to create a model that represents the many-to-many relationship as two one-to-many associations.

Solution

Entity Framework does not support associations with properties, so creating a model like the one in the previous recipe won't work. As we saw in the previous recipe, if the link table in a many-to-many relationship contains just the foreign keys for the relationship, Entity Framework will surface the link table as an association and not as an entity type. If the link table contains additional information, Entity Framework will create a separate entity type to represent the link table. The resulting model will contain two one-to-many associations with an entity type representing the underlying link table.

Suppose we have the tables and relationships shown in Figure 2-12.

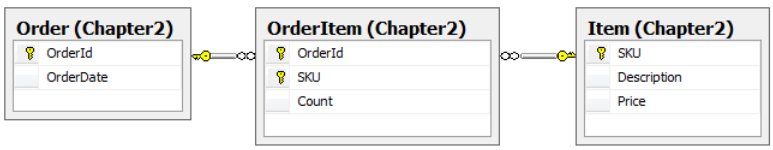


Figure 2-12. *A many-to-many relationship with payload*

An Order can have many Items. An Item can be on many orders. Additionally, we have a Count property connected to each instance of the Order, Item relationship. This Count property is referred to as a *payload*.

To create a model and import these tables and relationships into the model, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. From the Choose Your Database Object dialog box, select the tables Order, OrderItem, and Item. Leave the Pluralize and Foreign Key options checked. Click Finish.

The wizard will create the model in Figure 2-13.

Figure 2-13. *Two one-to-many associations from a many-to-many relationship with payload*

How It Works

As we saw in the previous recipe, for a many-to-many relationship with no payload, the model is clean and simple to navigate. Because Entity Framework does not support the notion of payloads on associations, it surfaces the link table as an entity with two one-to-many associations to the related entities. In this case, the OrderItem table is represented not as an association, but as an entity type with a one-to-many association to Order and a one-to-many association to Item. In the previous recipe, the payload-free link table did not translate into an entity type in the model. Instead, it became part of the many-to-many association.

The addition of a payload requires an additional hop through the entity representing the link table to retrieve the related items. This is illustrated in code in Listing 2-4.

Listing 2-4. Inserting into and retrieving from the model

```
using (var context = new EFRecipesEntities())
{
    var order = new Order { OrderId = 1,
                            OrderDate = new DateTime(2010, 1, 18) };
    var item = new Item { SKU = 1729, Description = "Backpack",
                          Price = 29.97M };
    var oi = new OrderItem { Order = order, Item = item, Count = 1 };
    item = new Item { SKU = 2929, Description = "Water Filter",
                      Price = 13.97M };
    oi = new OrderItem { Order = order, Item = item, Count = 3 };
    item = new Item { SKU = 1847, Description = "Camp Stove",
                      Price = 43.99M };
    oi = new OrderItem { Order = order, Item = item, Count = 1 };
    context.Orders.AddObject(order);
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    foreach (var order in context.Orders)
    {
        Console.WriteLine("Order # {0}, ordered on {1}",
                          order.OrderId.ToString(),
                          order.OrderDate.ToShortDateString());
        Console.WriteLine("SKU\tDescription\tQty\tPrice");
        Console.WriteLine("---\t-----------\t---\t-----");
        foreach (var oi in order.OrderItems)
        {
            Console.WriteLine("{0}\t{1}\t{2}\t{3}", oi.Item.SKU,
                              oi.Item.Description, oi.Count.ToString(),
                              oi.Item.Price.ToString("C"));
        }
    }
}
```

The following is the output from the code shown in Listing 2-4.

```
Order # 1, ordered on 1/18/2010

SKU     Description     Qty     Price

---     -----------     ---     -----

1729    Backpack        1       $29.97

1847    Camp Stove      1       $43.99

2929    Water Filter    3       $13.97
```

After we create the an instance of our object context, we create and initialize an order entity as well as the items and order items for the order. We connect the order with the items by initializing the OrderItem entities with the instances of the Order entity and the Item entity. We use the **AddToOrders()** method to add the order to the context.

With the object graph complete and the order added to the context, we update the database with the **SaveChanges()** method.

To retrieve the entities from the database, we create a fresh instance of the context and iterate through the context.Orders collection. For each order (well, we just have one in this example), we print the order detail and we iterate through the entity collection on the OrderItems navigation property. These instances of the OrderItem entity type give us access to the Count scalar property (the payload) directly and each item on the order via the Item navigation property. Going through the OrderItems entity to get to the items is the "extra" hop that is the cost of having a payload in the link table (OrderItems, in our example) in a many-to-many relationship.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Best Practice

Unfortunately, a project that starts out with several, payload-free, many-to-many relationships often ends up with several, payload-rich, many-to-many relationships. Refactoring a model, especially late in the development cycle, to accommodate payloads in the many-to-many relationships can be tedious. Not only are additional entities introduced, but the queries and navigation patterns through the relationships change as well. Some developers argue that every many-to-many relationship should start off with some payload, typically a synthetic key, so the inevitable addition of more payload has significantly less impact on the project.

So here's the best practice. If you have a payload-free, many-to-many relationship and you think there is some chance that it may change over time to include a payload, start with an extra identity column in the link table. When you import the tables into your model, you will get two one-to-many relationships, which means the code you write and the model you have will be ready for any number of additional payload columns that come along as the project matures. The cost of an additional integer identity column is usually a pretty small price to pay to keep the model more flexible.

2-5. Modeling a Self-Referencing Relationship

Problem

You have a table that references itself and you want to model this as an entity with a self-referencing association.

Solution

Let's say you have a self-referencing table that's like the one in the database diagram in Figure 2-14.

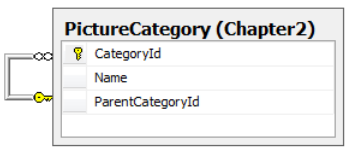


Figure 2-14. *A self-referencing table*

To create a model and import this table and the self-referencing relationship into the model, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the PictureCategory table. Leave the Pluralize and Foreign Key options checked. Click Finish.

The wizard will create a model like the one shown in Figure 2-15.

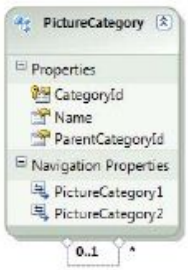


Figure 2-15. *Our initial model of a self-referencing PictureCategory table*

The model generated by the wizard contains two navigation properties named PictureCategory1 and PictureCategory2. Neither of these names is particularly helpful, so let's change them. One of these navigation properties refers to the parent category or the 0..1 side of the relationship. The other refers to the children or the * side of the relationship. To sort out which is which, right-click PictureCategory1. In the property window, the multiplicity for PictureCategory1 is * (many), so PictureCategory1 represents the navigation property for the children or subcategories. Rename PictureCategory1 to **Subcategories**. Change PictureCategory2 to **ParentCategory**.

The resulting model is shown in Figure 2-16.

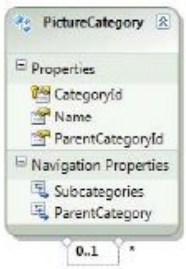


Figure 2-16. *The model with the correctly named navigation properties*

How It Works

Database relationships are characterized by degree, multiplicity, and direction. *Degree* is the number of entity types that participate in the relationship. Unary and binary relationships are the most common. Tertiary and n-place relationships are more theoretical than practical.

Multiplicity is the number of entity types on each end of the relationship. You have seen the multiplicities 0..1 (zero or 1), 1 (one), and * (many).

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Finally, the direction is either one-way or bi-directional.

The Entity Data Model supports a particular kind of database relationship called an Association Type. An Association Type relationship has either unary or binary degree, multiplicities 0..1, 1, or * and the direction is bi-directional.

In this example, the degree is unary (just the entity type PictureCategory is involved), the multiplicity is 0..1 and *, and the direction is, of course, bi-directional.

As is the case in this example, a self-referencing table often denotes a parent-child relationship with each parent having many children while each child has just one parent. Because the parent end of the relationship is 0..1 and not 1, it is possible for a child to have no parent. This is just what you want to leverage in representing the root node; that is, the one node that has no parent and is the top of the hierarchy.

Listing 2-5 shows how you can recursively enumerate the picture categories starting with the root node, which of course, is the only node that has no parent.

Listing 2-5. Inserting into our model and recursively enumerating all the instances of the self-referencing entity

```
static void RunExample()
{
    using (var context = new EFRecipesEntities())
    {
        var louvre = new PictureCategory { Name = "Louvre" };
        var child = new PictureCategory { Name = "Egyptian Antiquites" };
        louvre.Subcategories.Add(child);
        child = new PictureCategory { Name = "Sculptures" };
        louvre.Subcategories.Add(child);
        child = new PictureCategory { Name = "Paintings" };
        louvre.Subcategories.Add(child);
        var paris = new PictureCategory { Name = "Paris" };
        paris.Subcategories.Add(louvre);
        var vacation = new PictureCategory { Name = "Summer Vacation" };
        vacation.Subcategories.Add(paris);
        context.PictureCategories.AddObject(paris);
        context.SaveChanges();
    }

    using (var context = new EFRecipesEntities())
    {
        PictureCategory root = (from c in context.PictureCategories
                                where c.ParentCategory == null
                                select c).FirstOrDefault();
        Print(root, 0);
    }
}

static void Print(PictureCategory cat, int level)
{
    StringBuilder sb = new StringBuilder();
    Console.WriteLine("{0}{1}", sb.Append(' ', level).ToString(), cat.Name);
    foreach (PictureCategory child in cat.Subcategories)
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        Print(child, level + 1);
    }
}
```

The output of the code in Listing 2-5 shows our root node: Summer Vacation. The first (and only) child is Paris. Paris has Louvre as a child. And finally, at the Louvre, we categorized our pictures by the various collections we visited.

```
Summer Vacation

 Paris

  Louvre

   Egyptian Antiquites

   Sculptures

   Paintings
```

Okay, the code is a little involved. First, we create and initialize the instances of our entity types. We wire them together in the object graph by adding the PictureCategories to our louvre category. Then we add the louvre category to the paris category. Finally, we add the paris category to our summer vacation category. We build the hierarchy from the bottom up.

Once we do a `SaveChanges()`, the inserts are all done on the database, and it's time to query our tables to see whether we've actually inserted all the rows correctly.

For the retrieval part, we start by getting the root entity. This is the one that has no parent. In our case, we created a summer vacation entity, but we didn't make it the child of any other entity. This makes our summer vacation entity the root of the hierarchy.

Now with the root, we call another method we wrote: `Print()`. The `Print()` method takes a couple of parameters. The first parameter is an instance of a PictureCategory. The second parameter is a level, or depth we are at in the hierarchy. With the root category, summer vacation, we're at the top of the hierarchy, so we pass in 0. The method call looks like `Print(root, 0)`.

In the `Print()` method, we write out the name of the category preceded by a space for each level deep in the hierarchy. One of the `Append()` methods of the StringBuilder class takes a character and a integer count. It creates an instance of StringBuilder with the character appended count number of times. In our call, we send in a space and level and it returns a string with a space for every level deep we are in the hierarchy. We use the `ToString()` method to convert the StringBuilder instance to a string.

Now for the recursive part: we iterate through the children and call the `Print()` method on each child, making sure to increment the level by one. When we run out of children, we simply return. The result is the output shown previously.

In Recipe 6-5, we show another approach to this problem using a Common Table Expression in a stored procedure on the store side to iterate through the graph and return a single flattened result set.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

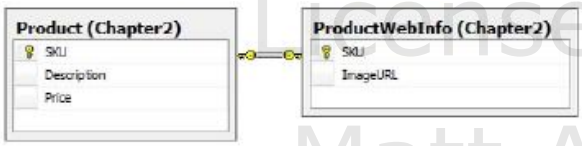
2-6. Splitting an Entity Across Multiple Tables

Problem

You have two or more tables that share the same primary key and you want to map a single entity to these two tables.

Solution

Let's illustrate the problem with the two tables shown in Figure 2-17.



Licensed by Matt Allen 2175288

Figure 2-17. *Two tables, Product and ProductWebInfo, with common primary keys*

To create a model with a single entity representing these two tables, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the Product and ProductWebInfo tables. Leave the Pluralize and Foreign Key options checked. Click Finish.

The resulting model is shown in Figure 2-18.

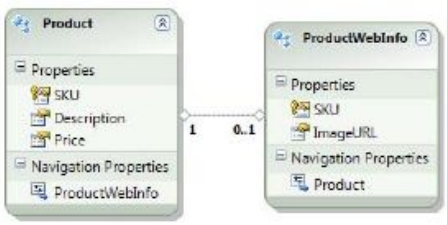


Figure 2-18. The model after importing the Product and ProductWebInfo tables

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Now that we have our table imported into the model, we need to merge the two entities into a single entity. To complete the model, do the following:

1. Copy the ImageURL scalar property from the ProductWebInfo entity to the Product entity. You can use copy/paste for this. Do not copy the SKU scalar property.
2. Right-click the ProductWebInfo entity and select Delete. The dialog box in Figure 2-19 will ask if you want to delete the tables from the store model. Select No. This will preserve the ProductWebInfo definition in the store model layer.
3. Click the Product entity to view the Mapping Details window. If the Mapping Details window is not visible, show it by selecting View ➤ Other Windows ➤ Entity Data Model Mapping Details.
4. In the Mapping Details window for the Product entity, click Add a Table or View and select the ProductWebInfo table. This adds the ProductWebInfo to the mappings for the Product entity.
5. Under the ProductWebInfo table in the Mapping Details window, map the ImageURL column to the ImageURL property. Also, make sure that the SKU property is mapped to the SKU column of the ProductWebInfo table. Your mappings should look like Figure 2-20.

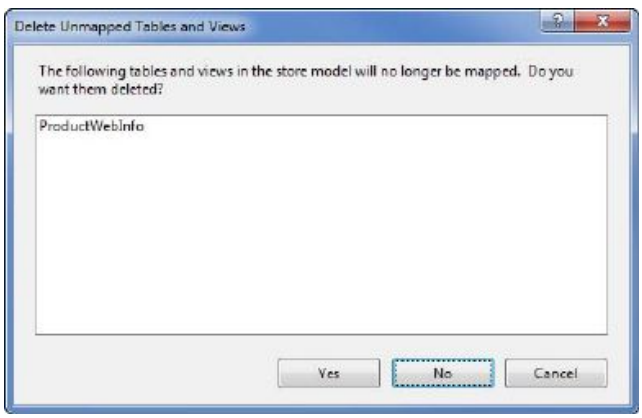


Figure 2-19. *A dialog box asking if the underlying ProductWebInfo table should be deleted from the store layer*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

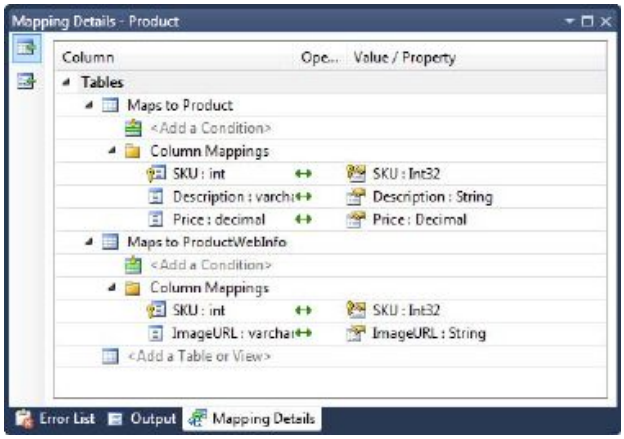


Figure 2-20. *Mappings Details window showing the mapping for the ProductWebInfo table in the Product entity. Notice that the entity maps to two tables with the SKU column, the key, mapped in both tables.*

The resulting model looks just like the one pictured in Figure 2-18, but without the ProductWebInfo entity type and the ImageURL property moved to the Product entity.

How It Works

It seems all too common in legacy systems to find "extra" information for each row in one table tucked away in another table. Often this happens over time as a database evolves and no one is willing to break existing code by adding columns to some critical table. The answer is to graft on a new table to hold the additional columns.

By merging two or more tables into a single entity, or as it is usually thought of, splitting a single entity across two or more tables, we can treat all the parts as one logical entity. This process is often referred to as *vertical splitting*.

The downside of vertical splitting is that retrieving each instance of our entity now requires an additional join for each additional table that makes up the entity type. This extra join is shown in Listing 2-6.

Listing 2-6. Additional join required by vertical splitting

```
SELECT
[Extent1].[SKU] AS [SKU],
[Extent2].[Description] AS [Description],
[Extent2].[Price] AS [Price],
[Extent1].[ImageURL] AS [ImageURL]
FROM  [dbo].[ProductWebInfo] AS [Extent1]
INNER JOIN [dbo].[Product] AS [Extent2] ON [Extent1].[SKU] = [Extent2].[SKU]
```

Nothing special is required to insert into or retrieve from the Product entity. Listing 2-7 demonstrates working with the vertically split Product entity type.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 2-7. Inserting into and retrieving from our model with the Product entity type

```
using (var context = new EFRecipesEntities())
{
    var product = new Product { SKU = 147,
                                Description = "Expandable Hydration Pack",
                                Price = 19.97M, ImageURL = "/pack147.jpg" };
    context.Products.AddObject(product);
    product = new Product { SKU = 178,
                            Description = "Rugged Ranger Duffel Bag",
                            Price = 39.97M, ImageURL = "/pack178.jpg" };
    context.Products.AddObject(product);
    product = new Product { SKU = 186,
                            Description = "Range Field Pack",
                            Price = 98.97M, ImageURL = "/noimage.jp" };
    context.Products.AddObject(product);
    product = new Product { SKU = 202,
                            Description = "Small Deployment Back Pack",
                            Price = 29.97M, ImageURL = "/pack202.jpg" };
    context.Products.AddObject(product);

    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var p in context.Products)
    {
        Console.WriteLine("{0} {1} {2} {3}", p.SKU, p.Description,
                            p.Price.ToString("C"), p.ImageURL);
    }
}
```

The code in Listing 2-7 produces the following results:

```
147 Expandable Hydration Pack $19.97 /pack147.jpg

178 Rugged Ranger Duffel Bag $39.97 /pack178.jpg

186 Range Field Pack $98.97 /noimage.jpg

202 Small Deployment Back Pack $29.97 /pack202.jpg
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

2-7. Splitting a Table Across Multiple Entities

Problem

You have a table with some frequently used fields and a few large but rarely needed fields. For performance reasons, you want to avoid needlessly loading these expensive fields on every query. You want to split the table across two or more entities.

Solution

Let's say you have a table like the one shown in Figure 2-21, which holds information about photographs as well as the bits for both the thumbnail and the full-resolution image of the photograph.

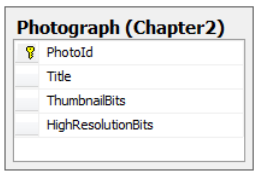


Figure 2-21. *A Photograph table with a field holding the binary large object (blob) representing the data for the image*

To create an entity type that contains the reasonably low cost and frequently used columns, as well as an entity type containing the high cost, rarely used HighResolutionBits column, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the Photograph table. Leave the Pluralize and Foreign Key options checked. Click Finish.
5. Right-click the design surface and select Add ➤ Entity. In the dialog box, change the Entity name to **PhotographFullImage**. Also, change the Key Property name to **PhotoId**. This is the same name as the key column in the Photograph entity type. See Figure 2-22. Click OK to add the PhotographFullImage entity type to the model.
6. Move the HighResolutionBits property from the Photograph entity type to the PhotographFullImage entity. You can use select/cut/paste to move the property.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

7. Click the newly created PhotographFullImage entity type to view the Mapping Details window. If the Mapping Details window is not visible, show it by selecting View ➤ Other Windows ➤ Entity Data Model Mapping Details.
8. In the Mapping Details window for the PhotographFullImage entity, click Add a Table or View and select the Photograph table. Map the HighResolutionBits column to the HighResolutionBits property. Map the PhotoId column to the PhotoId property. See Figure 2-23.
9. Right-click the Photograph entity. Select Add ➤ Association. Add a one-to-one association between the Photograph entity and the PhotographFullImage entity. Make sure you do two things: choose 1 (one) for the multiplicity on both ends of the association and uncheck the Add foreign key properties to the Entity check box. See Figure 2-24.
10. Right-click the association link between the entities and select Properties. In the Referential Constraint under the Constraints section, click the … button to add a referential constraint. Set the Principal to Photograph and make sure the key properties are set to PhotoId for both the principal and dependent entities. See Figure 2-25.

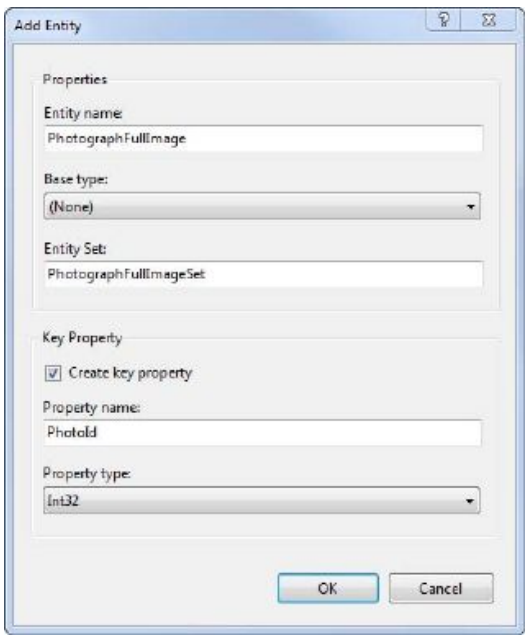


Figure 2-22. *Adding the PhotographFullImage entity to the Entity Data Model*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

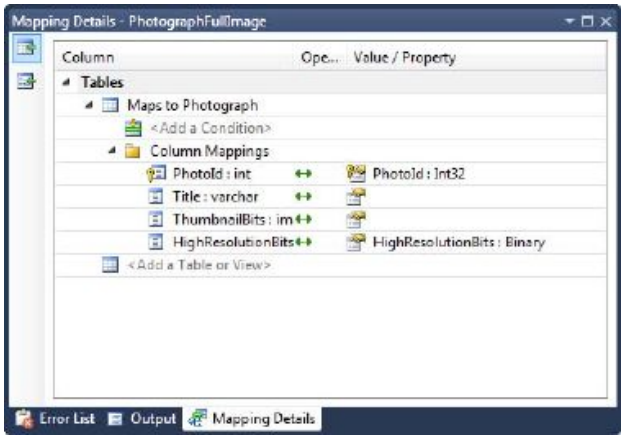


Figure 2-23. *The mapping details for the PhotographFullImage entity. We mapped the PhotoId and HighResolutionBits columns from the Photograph table to the respective properties on the PhotographFullImage entity.*

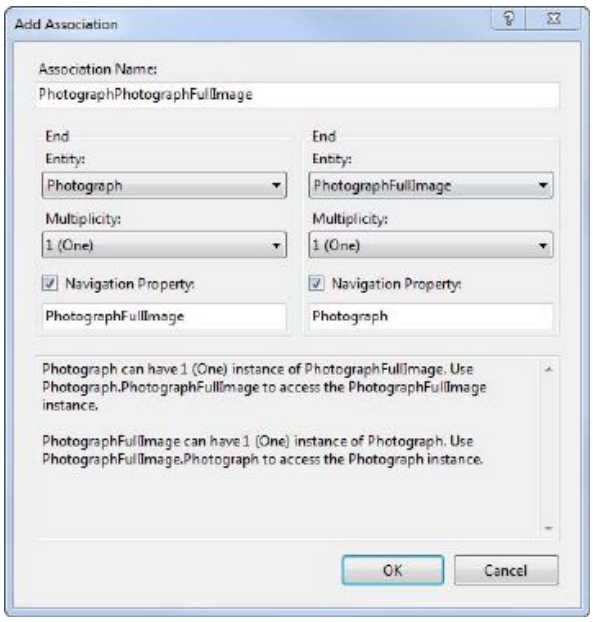


Figure 2-24. *Adding a one-to-one association between the Photograph entity type and the PhotographFullImage entity type*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

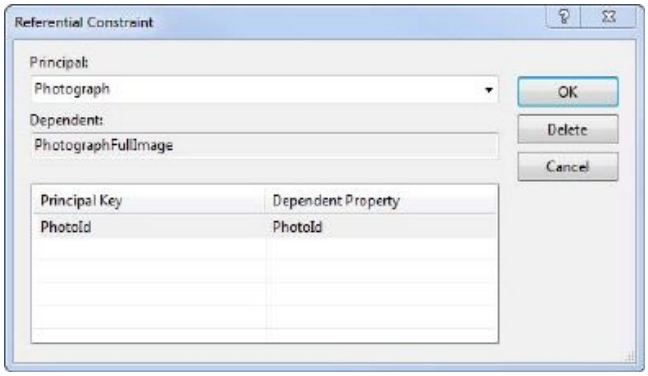


Figure 2-25. *Creating the Referential Constraint between the Photograph (principal) entity type and the PhotographFullImage (dependent) entity type*

The completed model is shown in Figure 2-26.

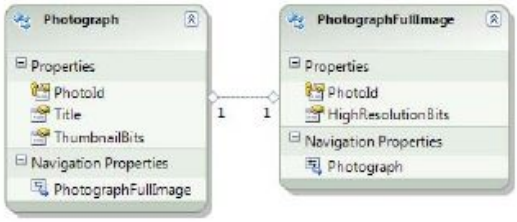


Figure 2-26. *The completed model with the HighResolutionBits column represented in a separate entity*

How It Works

Entity Framework does not directly support the notion of lazy loading of individual entity properties. To get the effect of lazy loading expensive properties, we exploit Entity Framework's support for lazy loading of associated entities. We created a new entity type to hold the expensive full image property and created a one-to-one association between our Photograph entity type and the new PhotographFullImage entity type. We added a referential constraint on the conceptual layer that, much like a database referential constraint, tells Entity Framework that a PhotographFullImage can't exist without a Photograph.

Because of the referential constraint, there are a couple of things to note about our model. If we have a newly created PhotographFullImage, an instance of Photograph must exist in the object context or the data source prior to calling `SaveChanges()`. Also, if we delete a Photograph, the associated PhotographFullImage is also deleted. This is just like cascading deletes in database referential constraints.

The code in Listing 2-8 demonstrates inserting and retrieving from our model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 2-8. Inserting into and Lazy Loading Expensive Fields

```
byte[] thumbBits = new byte[100];
byte[] fullBits = new byte[2000];
using (var context = new EFRecipesEntities())
{
    var photo = new Photograph { PhotoId = 1, Title = "My Dog",
                                 ThumbnailBits = thumbBits };
    var fullImage = new PhotographFullImage { PhotoId = 1,
                                 HighResolutionBits = fullBits };
    photo.PhotographFullImage = fullImage;
    context.Photographs.AddObject(photo);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var photo in context.Photographs)
    {
        Console.WriteLine("Photo: {0}, ThumbnailSize {1} bytes",
                          photo.Title, photo.ThumbnailBits.Length.ToString());

        // explicitly load the "expensive" entity, PhotographFullImage
        photo.PhotographFullImageReference.Load();
        Console.WriteLine("Full Image Size: {0} bytes",
                photo.PhotographFullImage.HighResolutionBits.Length.ToString());
    }
}
```

The output from Listing 2-8 is the following:

```
Photo: My Dog, Thumbnail Size: 100 bytes

Full Image Size: 2000 bytes
```

The code in Listing 2-8 creates and initializes instances of the Photograph and PhotographFullImage entities, adds them to the object context, and calls **SaveChanges()**.

On the query side, we retrieve each of the photographs from the database, print some information about the photograph, and then explicitly load the associated PhotographFullImage entity. Notice that we did not change the default context option that turns off lazy loading. This puts the burden on us to explicitly load related entities. This is just what we want. We could have chosen not to load the associated instances of PhotographFullImage, and if we were iterating through hundreds or thousands of photographs, this would have saved us an awful lot of cycles and bandwidth.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

2-8. Modeling Table per Type Inheritance

Problem

You have some tables that contain additional information about a common table and you want to model this using table per type inheritance.

Solution

Suppose you have two tables that are closely related to a common table as in Figure 2-27. The Business table is on the 1 side of a 1:0..1 relationship with the eCommerce and the Retail tables. The key feature here is that the eCommerce and Retail tables extend information about a business represented in the Business table.

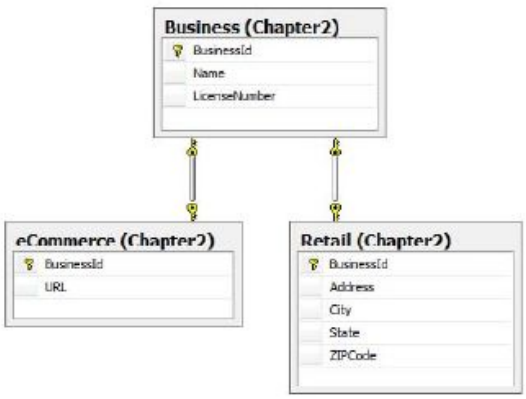


Figure 2-27. *Closely related tables ripe for inheritance*

The tables Retail and eCommerce are related to the Business table which holds a few properties we would naturally associate with any business. To model table per type inheritance such that entities Retail and eCommerce inherit from the Business base entity type, perform the following steps:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the Business, eCommerce, and Retail tables. Leave the Pluralize and Foreign Key options checked. Click Finish.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5. Delete the associations between the Retail and Business entities and between the eCommerce and Business entities.
6. Right-click the Business entity and choose Add ➤ Inheritance. In the dialog box, select Business as the base entity and Retail as the derived entity. Repeat this step for the eCommerce entity, setting eCommerce as an entity derived from the Business entity. See Figure 2-28.
7. Delete the BusinessId property from the Retail and eCommerce entities. For these entities, BusinessId will come from the Business entity.
8. Click the eCommerce entity to view the Mapping Details window. If the Mapping Details window is not visible, show it by selecting View ➤ Other Windows ➤ Entity Data Model Mapping Details. Map the BusinessId column to the BusinessId property. Repeat this step for the Retail entity. See Figure 2-29.

Licensed by Matt Allen 2175280

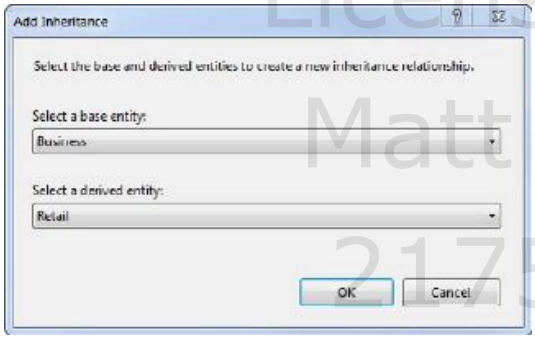


Figure 2-28. *Adding Inheritance between the Retail entity type and the Business entity type*

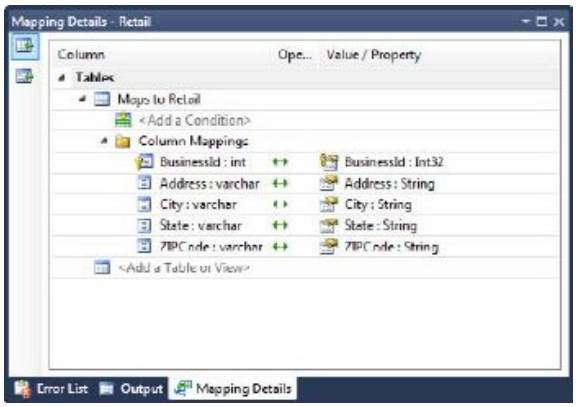


Figure 2-29. *Mapping the BusinessId column to the BusinessId property. This must be done for both the eCommerce entity type and the Retail entity type.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The resulting model is shown in Figure 2-30.

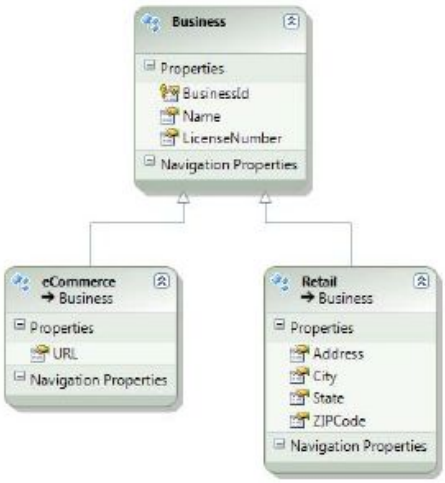


Figure 2-30. *Table per type inheritance with Retail and eCommerce deriving from the base entity type Business*

How It Works

Both the Retail and the eCommerce tables are on the 0..1 side of a 1:0..1 relationship with the Business table. This means that we could have a business with no additional information or a business with additional Retail or eCommerce information. In object-oriented programming terms, we have a base type, Business, with two derived types, Retail and eCommerce.

Because of the 1:0..1 relationship, we cannot have a row in the Retail or eCommerce tables without a corresponding row in the Business table. In object-oriented terms, an instance of a derived type has the properties of the base type. This concept of a derived type extending the properties of a base type is a key feature of inheritance. In table per type inheritance (often abbreviated TPT), each of the derived types is represented in separate tables.

Listing 2-9 demonstrates inserting and retrieving from our model.

Listing 2-9. Inserting and retrieving entities in TPT inheritance

```
using (var context = new EFRecipesEntities())
{
    var business = new Business { Name = "Corner Dry Cleaning",
                                  LicenseNumber = "100x1" };
    context.Businesses.AddObject(business);
    var retail = new Retail { Name = "Shop and Save", LicenseNumber = "200C",
                              Address = "101 Main", City = "Anytown",
                              State = "TX", ZIPCode = "76106" };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.Businesses.AddObject(retail);
    var web = new eCommerce { Name = "BuyNow.com", LicenseNumber = "300AB",
                              URL = "www.buynow.com" };
    context.Businesses.AddObject(web);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("\n--- All Businesses ---");
    foreach (var b in context.Businesses)
    {
        Console.WriteLine("{0} (#{1})", b.Name, b.LicenseNumber);
    }

    Console.WriteLine("\n--- Retail Businesses ---");
    foreach (var r in context.Businesses.OfType<Retail>())
    {
        Console.WriteLine("{0} (#{1})", r.Name, r.LicenseNumber);
        Console.WriteLine("{0}", r.Address);
        Console.WriteLine("{0}, {1} {2}", r.City, r.State, r.ZIPCode);
    }

    Console.WriteLine("\n--- eCommerce Businesses ---");
    foreach (var e in context.Businesses.OfType<eCommerce>())
    {
        Console.WriteLine("{0} (#{1})", e.Name, e.LicenseNumber);
        Console.WriteLine("Online address is: {0}", e.URL);
    }
}
```

The code in Listing 2-9 creates and initializes instances of the Business entity type and the two derived types. To add these to the Object Context, we use **AddObject()** method exposed on the Business entity set in the context.

On the query side, to access all the businesses, we iterate through the Businesses entity set. For the derived types, we use the **OfType<>()** method specifying the derived type to filter the Business entity set.

The output of Listing 2-9 looks like the following:

```
--- All Businesses ---

Corner Dry Cleaning (#100X1)

Shop and Save (#200C)

BuyNow.com (#300AB)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
--- Retail Businesses ---

Shop and Save (#200C)

101 Main

Anytown, TX 76106


---- eCommerce Businesses ---

BuyNow.com (#300AB)

Online address is: www.buynow.com
```

Table per type is one of three inheritance models supported by Entity Framework. The other two are Table per Hierarchy (discussed in this chapter) and Table per Concrete Type (see Chapter 15).

Table per type inheritance provides a lot of database flexibility because we can easily add tables as new derived types find their way into our model as an application develops. However, each derived type involves additional joins that can reduce performance. In real-world applications, we have seen significant performance problems with TPT when many derived types are modeled.

Table per hierarchy, as you will see in Recipe 2-10, stores the entire hierarchy in a single table. This eliminates the joins of TPT and thereby providing better performance but at the cost of some flexibility.

Table per concrete type is supported by the Entity Framework runtime, but not by the designer. Table per concrete type has some important applications, as we will see in Chapter 15.

2-9. Using Conditions to Filter an ObjectSet

Problem

You want to create a permanent filter on an entity type so that it maps to a subset of the rows in a table.

Solution

Let's say you have a table holding account information, as shown in the database diagram in Figure 2-31. The table has a DeletedOn nullable column that holds the date and time the account was deleted. If the account is still active, the DeletedOn column is null. We want our Account entity to represent only active accounts (i.e., account without a DeletedOn value).

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

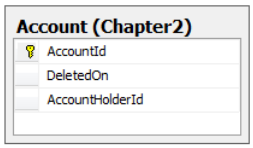


Figure 2-31. *Account table with DeletedOn DateTime column*

To model this table so that only active accounts are used to populate the Account entity type, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the Account table. Leave the Pluralize and Foreign Key options checked. Click Finish.
5. Click the Account entity to view the Mapping Details window. If the Mapping Details window is not visible, show it by selecting View ➤ Other Windows ➤ Entity Data Model Mapping Details. Click Add a Condition and select the DeletedOn column. In the Operator column, select Is; in the Value/Property column, select Null. This creates a mapping condition when the DeletedOn column is Is Null. See Figure 2-32.
6. Right-click the DeletedOn property and select Delete. Because we're using the DeletedOn column in a conditional mapping, we can't map it to a property. Its value would always be null anyway in our model.

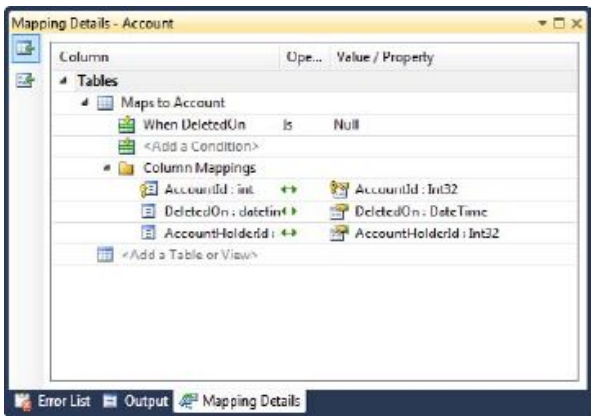


Figure 2-32. *Creating the conditional mapping for the Account entity to the Account table*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Conditional mappings are often used when you want to apply a permanent filter on an entity. Conditional mappings are also key to implementing Table per Hierarchy Inheritance. You can apply conditions using the following:

<value> **Is Null**

<value> **Is Not Null**

<integer> = <value>

<string> = <value>

In this example, we applied an **Is Null** condition on the Account entity that filters out rows that contain a DeletedOn date/time. The code in Listing 2-10 demonstrates inserting into and retrieving rows from the Account table.

Listing 2-10. Inserting into and retrieving from the account

```
using (var context = new EFRecipesEntities())
{
    context.ExecuteStoreCommand(@"insert into chapter2.account
            (DeletedOn,AccountHolderId) values ('2/10/2009',1728)");

    var account = new Account { AccountHolderId = 2320 };
    context.Accounts.AddObject(account);
    account = new Account { AccountHolderId = 2502 };
    context.Accounts.AddObject(account);
    account = new Account { AccountHolderId = 2603 };
    context.Accounts.AddObject(account);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var account in context.Accounts)
    {
        Console.WriteLine("Account Id = {0}",
                          account.AccountHolderId.ToString());
    }
}
```

In Listing 2-10, we use the `ExecuteStoreCommand()` method on the Object Context to insert a row into the database the old-fashioned way. We need to do this because we are inserting a row with a non-null value for the DeletedOn column. In our model, the Account entity type has no property mapping to this column; in fact, the Account entity type would never be materialized with a row that had a DeletedOn value. And that's exactly what we want to test.

The rest of the first part of the code creates and initializes three additional instances of the Account entity type. These are saved to the database with the `SaveChanges()` method.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When we query the database, we should get only the three instances of the Account entity type that we added with the **SaveChanges()** method. The row that we added using the **ExecuteStoreCommand()** method should not be visible. The following output confirms it:

```
Account Id = 2320

Account Id = 2502

Account Id = 2603
```

2-10. Modeling Table per Hierarchy Inheritance

Problem

You have a table with a type or discrimination column that you use to determine what the data in a row represents in your application. You want to model this with table per hierarchy inheritance.

Solution

Let's say your table looks like the one in Figure 2-33. This Employee table contains rows for both hourly employees and salaried employees. The EmployeeType column is used to discriminate between the two types of rows. When EmployeeType is 1, the row represents a salaried or full-time employee. When the EmployeeType is 2, the row represents an hourly employee.

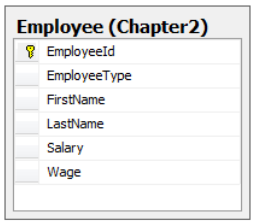


Figure 2-33. An Employee table containing both hourly and full-time employees

To create a model using table per hierarchy inheritance based on the Employee table, do the following:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. From the Choose Your Database Object dialog box, select the Employee table. Leave the Pluralize and Foreign Key options checked. Click Finish.
5. Right-click the design surface and select Add ➤ Entity. Name the new entity **FullTimeEmployee** and select Employee as the base type. Repeat this step creating a new **HourlyEmployee** entity deriving from Employee. See Figure 2-34.
6. Move the Salary property from the Employee entity to the FullTimeEmployee entity. You can use cut/paste to move the property. Using cut/paste, move the Wage property to the HourlyEmployee entity.
7. Click the FullTimeEmployee entity to view the Mapping Details window. If the Mapping Details window is not visible, show it by selecting View ➤ Other Windows ➤ Entity Data Model Mapping Details. Select the Employee table in the Add a Table or View control. Make sure that the Salary property is mapped to the Salary column.
8. In the Mapping Details window, add a condition by selecting EmployeeType in the Add a Condition control. Set the operator to = and the Value/Property to 1. This maps the Employee table to the FullTimeEmployee entity when the EmployeeType column is 1. See Figure 2-35.
9. Repeat steps 7 and 8 for the HourlyEmployee entity. Change the mapping condition to map the Employee table to the HourlyEmployee entity when the EmployeeType column has a value of 2.
10. Right-click the Employee entity and select Properties. Change the Abstract property to `true`. This makes the base entity abstract. In our model every employee must either be an hourly employee or a full-time employee.
11. Delete the EmployeeType property from the Employee entity.

The completed model is shown in Figure 2-36.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

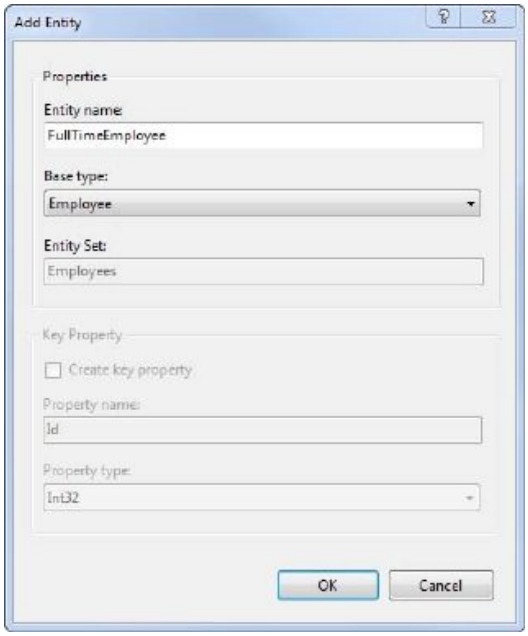


Figure 2-34. *Adding the FullTimeEmployee entity type that derives from Employee*

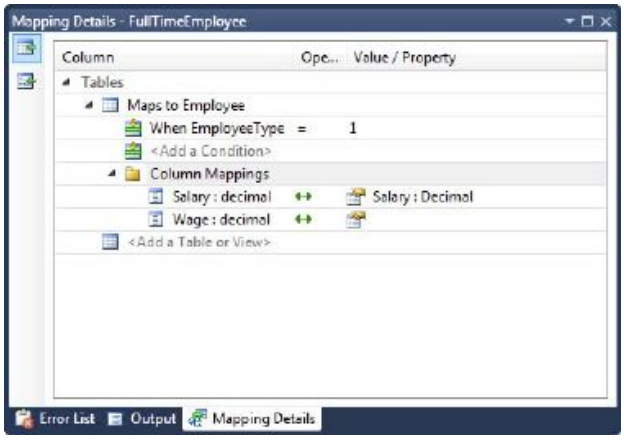


Figure 2-35. *Mapping the Employee table to the FullTimeEmployee entity type when the EmployeeType column has a value of 1*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

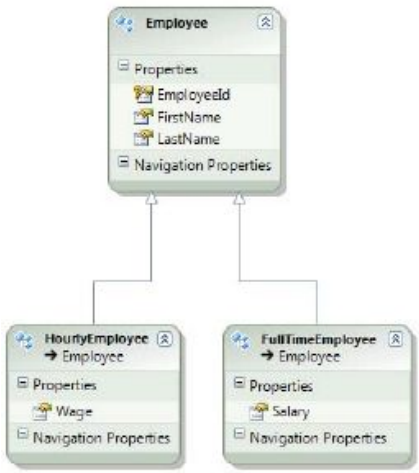


Figure 2-36. *The completed model with the HourlyEmployee and FullTimeEmployee entity types deriving from the abstract Employee entity type*

How It Works

In table per hierarchy inheritance, often abbreviated TPH, a single table is used to represent the entire inheritance hierarchy. Unlike table per type inheritance, in TPH rows for the derived types as well as the base type are intermingled in the same table. The rows are distinguished by a discriminator column. In our example, the discriminator column is EmployeeType.

In TPH, mapping conditions, available through the Mapping Details window, are used to indicate the values of the discrimination column that cause the table to be mapped to the different derived types. We marked the base type as abstract. By marking it as abstract, we didn't have to provide a condition for the mapping because an abstract entity can't be created. We will never have an instance of an Employee entity. We deleted the EmployeeType property from the Employee entity. A column used in a condition is not, in general, mapped to a property.

The code in Listing 2-11 demonstrates inserting into and retrieving from our model.

Listing 2-11. Inserting into and retrieving from our TPH model

```
using (var context = new EFRecipesEntities())
{
    var fte = new FullTimeEmployee { FirstName = "Jane", LastName = "Doe",
                                     Salary = 71500M};
    context.Employees.AddObject(fte);
    fte = new FullTimeEmployee { FirstName = "John", LastName = "Smith",
                                 Salary = 62500M };
    context.Employees.AddObject(fte);
    var hourly = new HourlyEmployee { FirstName = "Tom", LastName = "Jones",
                                      Wage = 8.75M };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.Employees.AddObject(hourly);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("--- All Employees ---");
    foreach (var emp in context.Employees)
    {
        bool fullTime = emp is HourlyEmployee ? false : true;
        Console.WriteLine("{0} {1} ({2})", emp.FirstName, emp.LastName,
                          fullTime ? "Full Time" : "Hourly");
    }

    Console.WriteLine("--- Full Time ---");
    foreach (var fte in context.Employees.OfType<FullTimeEmployee>())
    {
        Console.WriteLine("{0} {1}", fte.FirstName, fte.LastName);
    }

    Console.WriteLine("--- Hourly ---");
    foreach (var hourly in context.Employees.OfType<HourlyEmployee>())
    {
        Console.WriteLine("{0} {1}", hourly.FirstName, hourly.LastName);
    }
}
```

Licensed by Matt Allen 2175280

The following is the output of the code in Listing 2-11:

```
--- All Employees ---

Jane Doe (Full Time)

John Smith (Full Time)

Tom Jones (Hourly)

--- Full Time ---

Jane Doe

John Smith

--- Hourly ---

Tom Jones
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The code in Listing 2-11 creates, initializes, and adds two full-time employees and an hourly employee. On the query side, we retrieve all the employees and use the `is` operator to determine what type of employee we have. We indicate the employee type when we print out the employee's name.

In separate code blocks, we retrieve the full-time employees and the hourly employees using the `OfType<>()` method.

Best Practice

There is some debate over when to use abstract base entities in TPH inheritance and when to create a condition on the base entity. The difficulty with a concrete base entity is that it can be very cumbersome to query for all the instances in the hierarchy. The best practice is that if your application never needs instances of the base entity, make it abstract.

If your application needs instances of the base entity, consider introducing a new derived entity to cover the condition for the concrete base entity. For example, we might create a new derived class such as UnclassifiedEmployee. Once we have this new derived entity, we can safely make our base entity abstract. This provides us with a simple way to query for condition formally covered by the base entity with a condition.

There are some rules to keep in mind when using TPH. First, the conditions used must be mutually exclusive. That is, you cannot have a row that can conditionally map to two or more types.

Second, the conditions used must account for every row in the table. You cannot have a row in the table that has a discriminator value that does not map the row to exactly one type. This rule can be particularly troubling if you are working with a legacy database in which other applications are creating rows for which you have no appropriate condition mappings. What will happen in these cases? The rows that do no map to your base or derived types will simply not be accessible in your model.

The discriminator column cannot be mapped to an entity property unless it is used in an `is not null` condition. At first, this last rule might seem overly restrictive. You might ask, "How can I insert a row representing a derived type if I can't set the discriminator value?" The answer is rather elegant. You simply create an instance of the derived type and add it to the context in the same way you would any other entity instance. Object Services takes care of creating the appropriate insert statements to create a row with the correct discriminator value.

2-11. Modeling Is-a and Has-a Relationships Between Two Entities

Problem

You have two tables that participate in both Is-a and Has-a relations and you want to model them as two entities with the corresponding Is-a and Has-a relationships.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have two tables that describe scenic parks and their related locations. In your database, you represent these with a Location table and a Park table. For the purposes of your application, a park is simply a type of location. Additionally, a park can have a governing office with a mailing address, which is also represented in the Location table. A park then is both a derived type of Location and can have a location that corresponds to the park's governing office. It is entirely possible that the office is not located on the grounds of the park. Perhaps several parks share an office in a nearby town. Figure 2-37 shows a database diagram with the Park and Location tables.

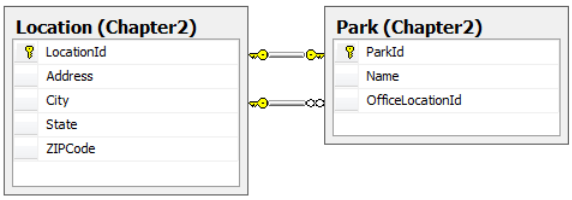


Figure 2-37. *Location and Park in both a Has-a and Is-a relationship*

Follow these steps to model both of these relationships:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the Location and Park tables. Leave the Pluralize and Foreign Key options checked. Click Finish.
5. Delete the one-to-zero or one association created by the Entity Data Model Wizard.
6. Right-click the Location entity and select Add ➤ Inheritance. Select the Park entity as the derived entity and the Location entity as the base entity.
7. Delete the ParkId property from the Park entity type.
8. Click the Park entity to view the Mapping Details window. If the Mapping Details window is not visible, show it by selecting View ➤ Other Windows ➤ Entity Data Model Mapping Details. Map the ParkId column to the LocationId property.
9. Change the name of the Location1 navigation property in the Park entity type to Office. This represents the office location for the park.

The completed model is shown in Figure 2-38.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

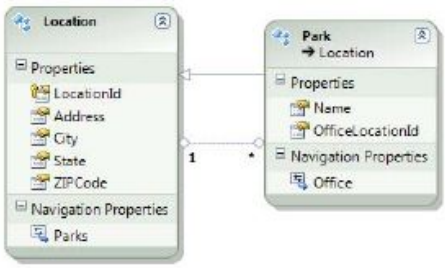


Figure 2-38. *The completed model with Park deriving from Location. A Park is-a location. A park has-a location for its office.*

How It Works

Entities can have more than one association with other entities. In this example, we created an Is-a relationship using table per type inheritance with Location as the base entity type and Park as the derived entity type. We also created a Has-a relationship with a one-to-many association between the Location and Park entity types.

In Listing 2-12, we demonstrate creating a new Park entity which also results in creating a Location because of the Is-a relationship. We attach an office Location to the Park, which results in a second row in the Location table.

Listing 2-12. Creating and retrieving Park and Location entities

```
using (var context = new EFRecipesEntities())
{
    var park = new Park { Name = "11th Street Park",
                          Address = "801 11th Street", City = "Aledo",
                          State = "TX", ZIPCode = "76106" };
    var loc = new Location { Address = "501 Main", City = "Weatherford",
                             State = "TX", ZIPCode = "76201" };
    park.Office = loc;
    context.Locations.AddObject(park);
    park = new Park { Name = "Overland Park", Address = "101 High Drive",
                      City = "Springtown", State = "TX", ZIPCode = "76081" };
    loc = new Location { Address = "8705 Range Lane", City = "Springtown",
                         State = "TX", ZIPCode = "76081" };
    park.Office = loc;
    context.Locations.AddObject(park);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    Console.WriteLine("-- All Locations -- ");
    foreach (var l in context.Locations)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    {
        Console.WriteLine("{0}, {1}, {2} {3}", l.Address, l.City,
                          l.State, l.ZIPCode);
    }

    Console.WriteLine("--- Parks ---");
    foreach (var p in context.Locations.OfType<Park>())
    {
        Console.WriteLine("{0} is at {1} in {2}", p.Name, p.Address, p.City);
        Console.WriteLine("\tOffice: {0}, {1}, {2} {3}", p.Office.Address,
                          p.Office.City, p.Office.State, p.Office.ZIPCode);
    }
}
```

The output from the code in Listing 2-12 is the following:

```
-- All Locations --

501 Main, Weatherford, TX 76201

801 11th Street, Aledo, TX 76106

8705 Range Lane, Springtown, TX 76081

101 High Drive, Springtown, TX 76081

--- Parks ---

11th Street Park is at 801 11th Street in Aledo

        Office: 501 Main, Weatherford, TX 76201

Overland Park is at 101 High Drive in Springtown

        Office: 8705 Range Lane, Springtown, TX 76081
```

2-12. Creating, Modifying, and Mapping Complex Types

Problem

You want to create a complex type, set it as a property on an entity, and map the property to some columns on a table.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have the table shown in Figure 2-39. You want to create a Name complex type for the FirstName and LastName columns. You also want to create an Address complex type for the AddressLine1, AddressLine2, City, State, and ZIPCode columns. You want to use these complex types for properties in your model as shown in Figure 2-40.

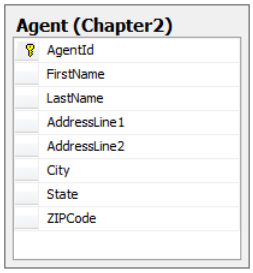


Figure 2-39. *The Agent table with the name and address of the agent*

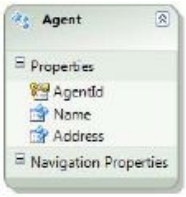


Figure 2-40. *The completed model with the name and address components refactored into complex types*

Follow these steps to create the model with the Name and Address complex types:

1. Add a new model to your project by right-clicking your project and selecting Add ➤ New Item. Choose ADO.NET Entity Data Model from the Visual C# Data templates.
2. Select Generate from database. Click Next.
3. Use the wizard to select an existing connection to your database or create a new connection.
4. From the Choose Your Database Object dialog box, select the Agent table. Leave the Pluralize and Foreign Key options checked. Click Finish.
5. Select the FirstName and LastName properties, right-click and select Refactor Into Complex Type.
6. In the Model Browser, rename the new complex type from ComplexType1 to Name. This changes the name of the type. On the Agent, rename the ComplexTypeProperty to Name. This changes the name of the property.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

7. We'll create the next complex type from scratch so you can see an alternate approach. Right-click on the design surface and select Add ➤ Complex Type.
8. In the Model Browser, rename the new complex type from ComplexType1 to Address.
9. Select the AddressLine1, AddressLine2, City, State, and ZIPCode properties in the Agent. Right-click and select Cut. Paste these properties onto the Address complex type in the Model Browser.
10. Right-click the Agent and select Add ➤ Complex Property. Rename the property Address.
11. Right-click on the new Address property and select Properties. Change its type to Address. This changes the new property's type to the new Address complex type.
12. View the Mapping Details window for the Agent. Map the columns from the Agent table to the properties on the two complex types we've created. The mappings are shown in Figure 2-41.

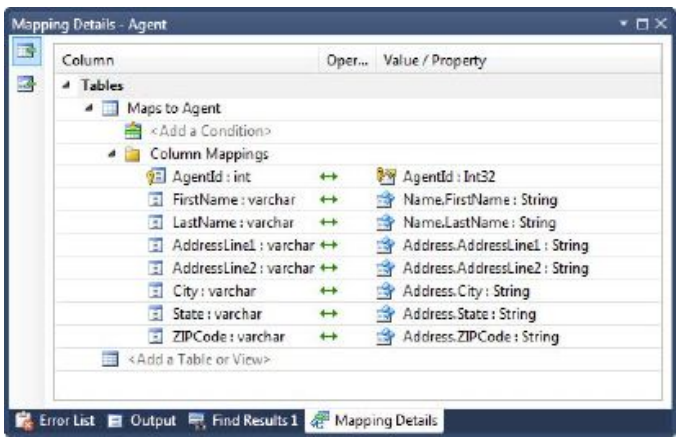


Figure 2-41. *Mapping the fields of the complex types to the columns in the Agent table*

How It Works

Complex types allow you to group several properties into a single type for a property on an entity. A complex type can contain scalar properties or other complex types, but they cannot have navigation properties or entity collections. A complex type cannot be an entity key. Complex types are not tracked on their own in an object context.

A property whose type is a complex type cannot be null. When you work with entities with complex type properties, you have to be mindful of this rule. Occasionally, when the value of a complex type property is unimportant for a particular operation, you may need to create a dummy value for the property so that it has some non-null value.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When you modify any field in complex type property, the property is marked as changed by Entity Framework and an update statement will be generated that will update all of the fields of the complex type property.

In Listing 2-13, we demonstrate using the model by inserting a few agents and displaying them.

Listing 2-13. Inserting agents and selecting from our model

```
using (var context = new EFRecipesEntities())
{
    var name1 = new Name { FirstName = "Robin", LastName = "Rosen" };
    var name2 = new Name { FirstName = "Alex", LastName = "St. James" };
    var address1 = new Address { AddressLine1 = "510 N. Grant",
                                 AddressLine2 = "Apt. 8",
                                 City = "Raytown", State = "MO",
                                 ZIPCode = "64133" };
    var address2 = new Address { AddressLine1 = "222 Baker St.",
                                 AddressLine2 = "Apt.22B",
                                 City = "Raytown", State = "MO",
                                 ZIPCode = "64133" };
    context.Agents.AddObject(new Agent { Name = name1, Address = address1 });
    context.Agents.AddObject(new Agent {Name = name2, Address = address2});
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Agents");
    foreach (var agent in context.Agents)
    {
        Console.WriteLine("{0} {1}", agent.Name.FirstName, agent.Name.LastName);
        Console.WriteLine("{0}", agent.Address.AddressLine1);
        Console.WriteLine("{0}", agent.Address.AddressLine2);
        Console.WriteLine("{0}, {1} {2}", agent.Address.City,
                          agent.Address.State, agent.Address.ZIPCode);
        Console.WriteLine();
    }
}
```

The output of the code in Listing 2-13 is the following:

```
Agents

Robin Rosen

510 N. Grant

Apt. 8

Raytown, MO 64133


Alex St. James
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
222 Baker St.
Apt.22B
Raytown, MO 64133
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 3

Querying an Entity Data Model

In the previous chapter, we showed you lots of different ways to model some fairly common database scenarios. In this chapter, we dive right into querying your models.

In these recipes we'll show you how to use LINQ and Entity SQL to query your models. We'll cover a wide range of common and some not so common scenarios that will help you understand some of the basics of querying your models.

Licensed by Matt Allen 2175280

3-1. Executing an SQL Statement

Problem

You want to execute an SQL statement.

Solution

Let's say you have a Payment table like the one shown in Figure 3-1 and you have created a model that looks like the one in Figure 3-2.

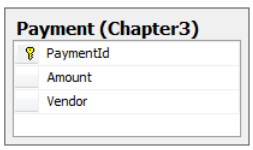


Figure 3-1. *A Payment table that contains information about a payment made by a vendor*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 3-2. *A model with a Payment entity type that was created when the model was updated with the Payment table*

You want to execute one or more SQL statements directly against the underlying Payment table. To do this, use the **ExecuteStoreCommand()** method available on the object context. Although we have a Payment entity in our model, it is not required. We simply need some model (after all, we need an object context) that is connected to the database against which we want to execute the SQL commands.

Follow the pattern in Listing 3-1 to execute one or more SQL statements.

Listing 3-1. Executing an Insert statement

```
// insert a couple rows
using (var context = new EFRecipesEntities())
{
    string sql = @"insert into Chapter3.Payment(Amount, Vendor)
                   values (@Amount, @Vendor)";
    var args = new DbParameter[] {
        new SqlParameter { ParameterName = "Amount", Value = 99.97M},
        new SqlParameter { ParameterName = "Vendor", Value="Ace Plumbing"}
    };
    int rowCount = context.ExecuteStoreCommand(sql, args);

    args = new DbParameter[] {
        new SqlParameter { ParameterName = "Amount", Value = 43.83M},
        new SqlParameter { ParameterName = "Vendor",
                           Value = "Joe's Trash Service"}
    };
    rowCount += context.ExecuteStoreCommand(sql, args);
    Console.WriteLine("{0} rows inserted", rowCount.ToString());
}

// materialize some entities
using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Payments");
    Console.WriteLine("========");
    foreach (var payment in context.Payments)
    {
        Console.WriteLine("Paid {0} to {1}", payment.Amount.ToString("C"),
                          payment.Vendor);
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following is the output of the code in Listing 3-1:

```
2 rows inserted

Payments

========

Paid $99.97 to Ace Plumbing

Paid $43.83 to Joe's Trash Service
```

How It Works

In Listing 3-1, we start off by creating a string with the SQL insert statement. This statement contains two parameters: @Amount and @Vendor. These are placeholders that will be replaced by values when the statement is executed.

Next, we create two parameters that bind the placeholder names to specific values. For the first insert, we bind the value 99.97 to the Amount placeholder. Next, we create a parameter that binds 'Ace Plumbing' to the Vendor placeholder.

To execute the SQL statement, we pass both the string with the SQL statement and the array of parameters to the `ExecuteStoreCommand()` method. `ExecuteStoreCommand()` returns the count of rows affected by the statement. In our case, one row is inserted each time we call `ExecuteStoreCommand()`.

If you don't have any parameters for a SQL statement, there is an overload of the `ExcuteStoreCommand()` method that takes just the SQL statement.

The pattern in Listing 3-1 is similar to how we would do the same thing in ADO.NET with SqlClient. The difference is that we don't need to construct a connection string and explicitly open a connection. This is handled by the object context.

The way we express the command text and the parameters are also different. With `ExecuteNonQuery()`, the command text and parameters are set on the underlying Command object. Here, these are passed into the `ExecuteStoreCommand()` method.

Of course, the observant reader will notice here that we're really not querying the model. In fact, as we mentioned, you don't need to have the Payment entity shown in Figure 3-2. The `ExecuteStoreCommand()` method simply uses the object context for its connection to the underlying data store.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Best Practice

To parameterize or not to parameterize, that is the question….Okay, Shakespeare aside, should I use parameters for SQL statements or just create the SQL statement strings that contain the parameters? The best practice is to use parameters whenever possible, and here are some reasons why:

- Parameterized SQL statements help prevent SQL Injection attacks. If you construct a complete SQL statement as a string by appending together strings that you get from a user interface such as an ASP.NET TextBox control, you may end up constructing a SQL statement that does some serious damage to your database or reveals some sensitive information. When you use parameterized SQL statements, the parameters are handled in a way that prevents this.
- Parameterized SQL statements, as we have shown in this recipe, allow you to reuse the non-varying part of the statement. This reuse can make your code more simple and easy to read.
- Parameterized SQL statements make your code more maintainable and configurable. For example, the statements could come from a configuration file. This would allow you to make some changes to the application without changing the code.

3-2. Returning Objects from a SQL Statement

Problem

You want to execute a SQL statement and get objects from your model.

Solution

Let's say you have a model with a Student entity type as shown in Figure 3-3.

Figure 3-3. *A model with a Student entity type*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

You want to execute a SQL statement that returns a collection of instances of the Student entity type. As we saw in the previous recipe, the **ExecuteStoreCommand()** method is similar to SQLCommand's **ExecuteNonQuery()** method. It executes the SQL statement for its side effects like inserting rows, and returns the number of rows affected. To materialize objects from our model, we can use the **ExecuteStoreQuery()** method on the object context.

To execute a SQL statement and get back a collection of instances of the Student entity type, follow the pattern in Listing 3-2.

*Listing 3-2. Using **ExecuteStoreQuery()** to execute a SQL statement and get back objects*

```
using (var context = new EFRecipesEntities())
{
    context.Students.AddObject(new Student { FirstName = "Robert",
                                LastName = "Smith", Degree = "Masters" });
    context.Students.AddObject(new Student { FirstName = "Julia",
                                LastName = "Kerns", Degree = "Masters" });
    context.Students.AddObject(new Student { FirstName = "Nancy",
                                LastName = "Stiles", Degree = "Doctorate" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    string sql = "select * from Chapter3.Student where Degree = @Major";
    var args = new DbParameter[] {
        new SqlParameter {ParameterName = "Major", Value = "Masters"}};
    var students = context.ExecuteStoreQuery<Student>(sql, args);
    Console.WriteLine("Students...");
    foreach (var student in students)
    {
        Console.WriteLine("{0} {1} is working on a {2} degree",
                    student.FirstName, student.LastName, student.Degree);
    }
}
```

The following is the output of the code in Listing 3-2:

```
Students...

Robert Smith is working on a Masters degree

Julia Kerns is working on a Masters degree
```

How It Works

In Listing 3-2, we add three Students to the object context and save them to the database using **SaveChanges()**.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

To retrieve the Students who are working on a Masters degree, we use the `ExecuteStoreQuery()` method with a parameterized SQL statement and a parameter set to "Masters." We iterate through the returned collection of Students and print each of them.

Here we use * in place of explicitly naming each column in the select statement. This works because the columns in the underlying table match the properties in the Student entity type. Entity Framework will match up the returned values to the appropriate properties. This works out fine in most cases, but if you have fewer columns returned from your query, Entity Framework will throw an exception during the materialization of the object. This can easily be fixed by adding dummy columns and values of the appropriate types to your query.

If your SQL statement returns more columns than required to materialize the entity, Entity Framework will happily ignore the additional columns.

There are some restrictions with the `ExecuteStoreQuery()` method. If you are using Table per Hierarchy inheritance and your SQL statement returns rows that could map to different derived types, Entity Framework will not be able to use the discriminator column to map the rows to the correct derived types. You will likely get a runtime exception because some rows don't contain the values required for the type being materialized.

If an entity has a complex type property, then instances of the entity can't be returned using `ExecuteStoreQuery()`. However, `ExecuteStoreQuery()` can be used to return a collection of instances of a complex type. Returning instances of a complex type, which is supported, is subtly different from returning instances of an entity that contains a complex type, which is not supported.

You can use `ExecuteStoreQuery()` to materialize objects that are not entities at all. For example, we could create a `StudentName` class that contains just first and last names of a student. If our SQL statement returned just these two strings, then we could use `ExecuteStoreQuery<StudentName>()` along with our SQL statement to get a collection of instances of `StudentName`.

We've been careful to use the phrase SQL statement rather than select statement because the `ExecuteStoreQuery()` method works with any SQL statement that returns a row set. This includes, of course, select statements, but also includes statements that execute stored procedures.

There is a version of `ExecuteStoreQuery()` that takes a parameter that determines how the returned objects are merged into the object context. By default, `ExecuteStoreQuery()` uses the `MegeOption.NoTracking`, which means that the returned objects are not tracked in the object context. If you happened to retrieve an object that has the same entity key as an object that is already in the object context, you will get a fresh copy of the object. If you use this version of the `ExecuteStoreQuery()` method, the second parameter is the name of the entity set that contains the entity.

3-3. Returning Objects from an Entity SQL Statement

Problem

You want to execute an Entity SQL statement that queries your model and returns objects.

Solution

Let's say you have a model like the one in Figure 3-4. The model contains a single Customer entity type. The Customer entity type has a Name and an Email property. You want to query this model using Entity SQL.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

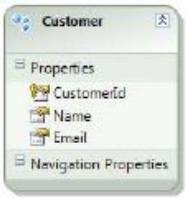


Figure 3-4. *A model with a Customer entity*

To query the model using Entity SQL, follow the pattern in Listing 3-3. The code in Listing 3-3 demonstrates executing and Entity SQL statement using both Object Services and EntityClient.

Listing 3-3. Executing an Entity SQL statement using both Object Services and EntityClient

```
using (var context = new EFRecipesEntities())
{
    var cus1 = new Customer { Name = "Robert Stevens",
                              Email = "rstevens@mymail.com" };
    var cus2 = new Customer { Name = "Julia Kerns",
                              Email = "julia.kerns@abc.com" };
    var cus3 = new Customer { Name = "Nancy Whitrock",
                              Email = "nrock@myworld.com" };
    context.Customers.AddObject(cus1);
    context.Customers.AddObject(cus2);
    context.Customers.AddObject(cus3);
    context.SaveChanges();
}

// using object services
using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Customers...");
    string esql = "select value c from Customers as c";
    var customers = context.CreateQuery<Customer>(esql);
    foreach (var customer in customers)
    {
        Console.WriteLine("{0}'s email is: {1}",
                          customer.Name, customer.Email);
    }
}

Console.WriteLine();

// using EntityClient
using (var conn = new EntityConnection("name=EFRecipesEntities"))
{
    Console.WriteLine("Customers...");
    var cmd = conn.CreateCommand();
    conn.Open();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    cmd.CommandText = "select value c from EFRecipesEntities.Customers as c";
    using (var reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
    {
        while (reader.Read())
        {
            Console.WriteLine("{0}'s email is: {1}",
                              reader.GetString(1), reader.GetString(2));
        }
    }
}
```

The following is the output from the code in Listing 3-3:

```
Customers...

Robert Stevens's email is: rstevens@mymail.com

Julia Kerns's email is: julia.kerns@abc.com

Nancy Whitrock's email is: nrock@myworld.com



Customers...

Robert Stevens's email is: rstevens@mymail.com

Julia Kerns's email is: julia.kerns@abc.com

Nancy Whitrock's email is: nrock@myworld.com
```

How It Works

In Listing 3-4, we create three customers, add them to the object context, then call **SaveChanges()** to save these new customers to the database.

After we have these customers in the database, we use two different approaches to retrieve them using Entity SQL. In the first approach, we use the **CreateQuery()** method on the object context to create an **ObjectQuery**. When we iterate over the customers, the query is executed in the database and the resulting collection is printed to the console. Because each element in the collection is an instance of our Customer entity type, we can use the properties of the Customer entity type.

In the second approach, we use EntityClient in a pattern that is very similar to how we would use SqlClient or any of the other client providers in ADO.NET. We start by creating a connection to the database. With the connection in hand, we create a command object and open the connection. Next we initialize the command object with the text of the Entity SQL statement we want to execute. We execute the command using **ExecuteReader()** and obtain an **EntityDataReader**, which is a type of the familiar **DbDataReader**. We iterate over the resulting collection using the **Read()** method.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The Entity SQL statement in Listing 3-3 uses the **value** keyword. This keyword is useful when we need the entire entity. If our Entity SQL statement forms a projection of the columns (that is, we use some of the columns and/or create columns using Entity SQL expressions), we can dispense with the **value** keyword. When using Object Services, this means working with a **DbDataRecord** directly. The code in Listing 3-4 demonstrates this.

Listing 3-4. Projecting with both Object Services and EntityClient

```
// using object services without the VALUE keyword
using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Customers...");
    string esql = "select c.Name, c.Email from Customers as c";
    var records = context.CreateQuery<DbDataRecord>(esql);
    foreach (var record in records)
    {
        var name = record[0] as string;
        var email = record[1] as string;
        Console.WriteLine("{0}'s email is: {1}", name, email);
    }
}

Console.WriteLine();

// using EntityClient without the VALUE keyword
using (var conn = new EntityConnection("name=EFRecipesEntities"))
{
    Console.WriteLine("Customers...");
    var cmd = conn.CreateCommand();
    conn.Open();
    cmd.CommandText = @"select c.Name, C.Email from
                        EFRecipesEntities.Customers as c";
    using (var reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
    {
        while (reader.Read())
        {
            Console.WriteLine("{0}'s email is: {1}",
                        reader.GetString(0), reader.GetString(1));
        }
    }
}
```

When you form a projection in Entity SQL, the results are returned in a **DbDataRecord** object that contains one element for each column in the projection. With the **value** keyword, the single object resulting from the query is returned in the first element of the **DbDataRecord**.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3-4. Specifying Fully Qualified Names in Entity SQL

Problem

You want to fully qualify an entity type with the correct namespace inside of an Entity SQL statement.

Solution

Let's say you have a simple model using Table per Type inheritance, as shown in Figure 3-5.

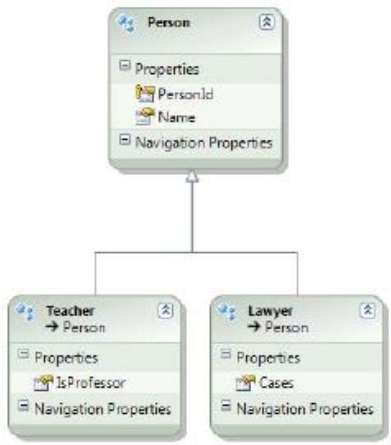


Figure 3-5. *A model using Table per Type inheritance with the derived entities Teacher and Lawyer*

In Figure 3-5, we have two entities, Teacher and Lawyer, which are derived from the Person entity type. Because we're using Table per Type inheritance, each of the derived types is represented in a separate table.

To query the model for all the Teachers using Object Services, we need to qualify the Teacher entity type with the Recipe4 namespace. This is the CLR namespace that contains our object context and our entities. Because we are interested in only the Teacher entities, we use the Entity SQL `OfType()` operator passing in the People entity set that contains our Teacher entity and the fully qualified Teacher entity. This is the first query in Listing 3-5.

The same query using EntityClient requires that we qualify the entity set with the Entity Container name EFRecipesEntities and the Teacher entity type with the namespace of the conceptual model, EFRecipesModel. This is shown in the second query in Listing 3-5.

Listing 3-5. Retrieving the teachers using Object Services and EntityClient

```
using (var context = new EFRecipesEntities())
{
    context.People.AddObject(new Teacher { Name = "Janet Dietz",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                        IsProfessor = true });
    context.People.AddObject(new Teacher { Name = "Robert Kline",
                                        IsProfessor = false });
    context.People.AddObject(new Lawyer { Name = "Jenny Dunlap", Cases = 3 });
    context.People.AddObject(new Lawyer { Name = "Karen Eads", Cases = 7 });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var esql = "select value p from OfType(People,Recipe4.Teacher) as p";
    var teachers = context.CreateQuery<Teacher>(esql);
    Console.WriteLine("Teachers...Using Object Services");
    foreach (var teacher in teachers)
    {
        Console.WriteLine("{0} is{1} a professor", teacher.Name,
                          teacher.IsProfessor ? "" : " not");
    }
}

Console.WriteLine();

using (var conn = new EntityConnection("name=EFRecipesEntities"))
{
    conn.Open();
    var esql = @"select value p from
                OfType(EFRecipesEntities.People,EFRecipesModel.Teacher) as p";
    var cmd = conn.CreateCommand();
    cmd.CommandText = esql;
    Console.WriteLine("Teachers...Using EntityClient");
    using (var reader = cmd.ExecuteReader(CommandBehavior.SequentialAccess))
    {
        while (reader.Read())
        {
            Console.WriteLine("{0} is{1} a professor", reader.GetString(1),
                              reader.GetBoolean(2) ? "" : " not");
        }
    }
}
```

The following is the output of the code in Listing 3-5:

```
Teachers...Using Object Services

Janet Dietz is a professor

Robert Kline is not a professor
```


Licensed by Matt Allen 2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Teachers...Using EntityClient

Janet Dietz is a professor

Robert Kline is not a professor
```

How It Works

In Listing 3-5, we referenced the People entity set in each query. Normally, when you reference an entity set, it is qualified by the Entity Container name. In our case, this is EFRecipesEntities. In the first query, we didn't need to fully qualify the entity set because we are executing the query in the object context and the entity set is in the default container for the context. In the second query, the one in which we execute against EntityClient, we need to fully qualify the entity set with the Entity Container name.

In the first query, we qualified the Teacher entity type with the CLR namespace in which the entity was generated. This namespace is, by default, the namespace of the project. You can change the namespace for the generated code in the Custom Tool Namespace property of the model. To change this namespace, right-click the .edmx file, select Properties and change the Custom Tool Namespace.

For the second query, we qualified the Teacher entity type with the namespace of the conceptual model. For the EntityClient approach, we are not using the generated code so we can't use the CLR namespace. To find the conceptual model namespace, right-click the model design surface and select Properties. The conceptual model namespace is listed under the Namespace property.

If you have a more complex query, you can make it somewhat more readable with the Entity SQL **using** clause. Like the using statement in C#, this clause allows you to factor out the qualifiers and simplify the code. We could rewrite the first Entity SQL statement as follows:

```
"using Recipe4; select value p from OfType(People,Teacher) as p";
```

3-5. Finding a Master that Has Detail in a Master-Detail Relationship

Problem

You have two entities in a one-to-many association (a.k.a. Master-Detail) and you want to find all the master entities that have at least one associated detail entity.

Solution

Imagine you have a model for blog posts and the comments associated with each post. Some posts have lots of comments. Some posts have few or no comments. The model might look something like the one in Figure 3-6.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

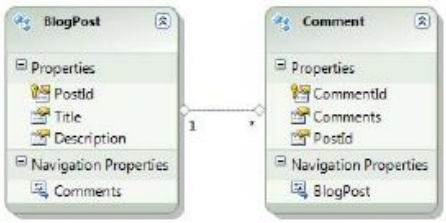


Figure 3-6. *A model for blog posts and the associated comments*

You want to find all the blog posts that have at least one comment. To do this using either LINQ to Entities or Entity SQL, follow the pattern in Listing 3-6.

Listing 3-6. Finding the masters that have detail using both LINQ and Entity SQL

```
using (var context = new EFRecipesEntities())
{
    var post1 = new BlogPost { Title = "The Joy of LINQ",
           Description = "101 things you always wanted to know about LINQ" };
    var post2 = new BlogPost { Title = "LINQ as Dinner Conversation",
           Description = "What wine goes with a Lambda expression?" };
    var post3 = new BlogPost {Title = "LINQ and our Children",
           Description = "Why we need to teach LINQ in High School"};
    var comment1 = new Comment {
         Comments = "Great post, I wish more people would talk about LINQ" };
    var comment2 = new Comment {
         Comments = "You're right, we should teach LINQ in high school!" };
    post1.Comments.Add(comment1);
    post3.Comments.Add(comment2);
    context.BlogPosts.AddObject(post1);
    context.BlogPosts.AddObject(post2);
    context.BlogPosts.AddObject(post3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Blog Posts with comments...(LINQ)");
    var posts = from post in context.BlogPosts
                where post.Comments.Any()
                select post;
    foreach (var post in posts)
    {
        Console.WriteLine("Blog Post: {0}", post.Title);
        foreach (var comment in post.Comments)
        {
            Console.WriteLine("\t{0}", comment.Comments);
        }
    }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
}

Console.WriteLine();

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Blog Posts with comments...(ESQL)");
    var esql = "select value p from BlogPosts as p where exists(p.Comments)";
    var posts = context.CreateQuery<BlogPost>(esql);
    foreach (var post in posts)
    {
        Console.WriteLine("Blog Post: {0}", post.Title);
        foreach (var comment in post.Comments)
        {
            Console.WriteLine("\t{0}", comment.Comments);
        }
    }
}
```

The following is the output of the code in Listing 3-6:

```
Blog Posts with comments...(LINQ)

Blog Post: The Joy of LINQ

        Great post, I wish more people would talk about LINQ

Blog Post: LINQ and our Children

        You're right, we should teach LINQ in high school!



Blog Posts with comments...(ESQL)

Blog Post: The Joy of LINQ

        Great post, I wish more people would talk about LINQ

Blog Post: LINQ and our Children

        You're right, we should teach LINQ in high school!
```

How It Works

We start off the code in Listing 3-6 by inserting a few blog posts and comments into the database. We left one of the blog posts without any comments to make sure our query is doing the right thing.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In the LINQ query, we use the **Any()** method in the **where** clause to determine whether there are comments for a given post. The query finds all the posts for which the **Any()** method returns **true**. And that's just what we want: all the posts for which there is at least one comment.

For the Entity SQL approach, we use the **exists()** operator, again in a **where** clause, to determine whether the given post has at least one comment.

There are, of course, other ways to get the same results. We could have used the **Count()** method in the LINQ query's **where** clause and tested if the count is greater than 0. For the Entity SQL approach, we could use **count(select value 1 from p.Comments) > 0** in the **where** clause. Either of these approaches would work. The code in Listing 3-6 seems a bit cleaner and, if it's any consolation, the semantics behind **Any()** and **exists()** don't require the enumeration of the entire collection on the server, whereas **count()** does require a full enumeration on the server.

3-6. Setting Default Values in a Query

Problem

You want to assign a default value to a property that has null value in a query.

Solution

Let's say you have a model like the one shown in Figure 3-7. You want to query the model for employees. In the database, the table representing employees contains a nullable YearsWorked column. This is the column mapped to the YearsWorked property in the Employee entity. You want the rows that contain a null value for the YearsWorked to default to the value 0.

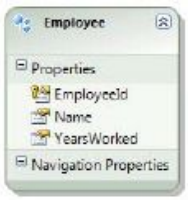


Figure 3-7. *A model with an Employee entity type containing an EmployeeId property, a Name property, and a YearsWorked property*

There are a couple of different approaches you can take here. The simplest is to change the Default Value property on YearsWorked to 0. To do this, right-click YearsWorked and select Properties. In the Properties window, change the Default Value to 0.

You can also assign default values via a query as in Listing 3-7. Note that the pattern in Listing 3-7 doesn't really materialize instances of the Employee entity type with the default value. Instead, it projects the results of the query into a collection of an anonymous type whose YearsWorked property is 0 whenever the underlying value is null.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 3-7. Using both LINQ and Entity SQL to fill in default values for nulls

```
using (var context = new EFRecipesEntities())
{
    context.Employees.AddObject(new Employee { Name = "Robin Rosen",
                                               YearsWorked = 3 });
    context.Employees.AddObject(new Employee { Name = "John Hancock" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Employees (using LINQ)");
    var employees = from e in context.Employees
                    select new {Name = e.Name, YearsWorked = e.YearsWorked ?? 0};
    foreach(var employee in employees)
    {
        Console.WriteLine("{0}, years worked: {1}",employee.Name,
                          employee.YearsWorked);
    }
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Employees (using ESQL)");
    string esql = @"select
                      e.Name,
                      case when e.YearsWorked is null then 0
                           else e.YearsWorked
                      end as YearsWorked
                    from Employees as e";
    var employees = context.CreateQuery<DbDataRecord>(esql);
    foreach (var employee in employees)
    {
        Console.WriteLine("{0}, years worked: {1}", employee.GetString(0),
                          employee.GetInt32(1).ToString());
    }
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Employees (using ESQL w/named constructor)");
    string esql = @"select value Recipe6.Employee(e.EmployeeId,
                      e.Name,
                      case when e.YearsWorked is null then 0
                           else e.YearsWorked end)
                    from Employees as e";
    var employees = context.CreateQuery<Employee>(esql);
    foreach(var employee in employees)
    {
        Console.WriteLine("{0}, years worked: {1}",employee.Name,
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                    employee.YearsWorked.ToString());
    }
}
```

The following is the output of the code in Listing 3-7:

```
Employees (using LINQ)

Robin Rosen, years worked: 3

John Hancock, years worked: 0

Employees (using ESQL)

Robin Rosen, years worked: 3

John Hancock, years worked: 0

Employees (using ESQL w/named constructor)

Robin Rosen, years worked: 3

John Hancock, years worked: 0
```

How It Works

As we mentioned, the simple solution is to set the Default Value to 0 for the YearsWorked property. This will cause the instances of the Employee entity type to be materialized with a 0 for the YearsWorked property when the underlying value is null.

The other approach is to use either LINQ or ESQL to project the results into a collection of an anonymous type. The query sets the YearsWorked to 0 when the underlying value is null.

For the LINQ approach, we use the null-coalescing operator ?? to assign the value of 0 when the underlying value is null. We project the results into a collection of an anonymous type.

For Entity SQL we use a case statement to assign the value of 0 to YearsWorked when the underlying value is null.

In the last bit of code, we show how to use Entity SQL to materialize instances of the Employee entity type without setting the Default Value property for the entity. To do this, we use the named constructor for the entity type. This constructor assigns the values from the parameters to the properties in the same order as the properties are defined in the entity. In our case, the properties for the Employee entity are defined in the following order: EmployeeId, Name, and YearsWorked. The parameters to the constructor follow this same order. We also changed the type for the **CreateQuery()** method from **DbDataRecord** to **Employee**.

Unfortunately, there is no corresponding name constructor syntax for LINQ to Entities.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3-7. Returning Multiple Result Sets From a Stored Procedure

Problem

You have a stored procedure that returns multiple result sets and you want to materialize entities from each result set.

Solution

Suppose you have a model like the one in Figure 3-8 and a stored procedure like the one in Listing 3-8 that returns both jobs and bids.

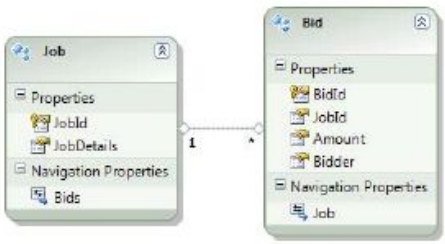


Figure 3-8. *A model representing jobs and bids for the jobs*

Listing 3-8. A stored procedure that returns multiple result sets

```
create procedure [Chapter3].[GetBidDetails]
as
begin
 select * from Chapter3.Job
 select * from Chapter3.Bid
end
```

In our model, for each job we have zero or more bids. Our stored procedure returns all the jobs and all the bids. We want to execute the stored procedure and materialize all the jobs and all the bids from the two result sets. To do this, follow the pattern in Listing 3-9.

Listing 3-9. Materializing jobs and bids from the two result sets returned by our stored procedure

```
using (var context = new EFRecipesEntities())
{
    var job1 = new Job { JobDetails = "Re-surface Parking Log" };
    var job2 = new Job { JobDetails = "Build Driveway" };
    job1.Bids.Add(new Bid { Amount = 948M, Bidder = "ABC Paving" });
    job1.Bids.Add(new Bid { Amount = 1028M, Bidder = "TopCoat Paving" });
    job2.Bids.Add(new Bid { Amount = 502M, Bidder = "Ace Concrete" });
    context.Jobs.AddObject(job1);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.Jobs.AddObject(job2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var cs = @"Data Source=.;Initial Catalog=EFRecipes;Integrated Security=True";
    var conn = new SqlConnection(cs);
    var cmd = conn.CreateCommand();
    cmd.CommandType = System.Data.CommandType.StoredProcedure;
    cmd.CommandText = "Chapter3.GetBidDetails";
    conn.Open();
    var reader = cmd.ExecuteReader(CommandBehavior.CloseConnection);
    var jobs = context.Translate<Job>(reader, "Jobs",
                                      MergeOption.AppendOnly).ToList();
    reader.NextResult();
    context.Translate<Bid>(reader, "Bids", MergeOption.AppendOnly).ToList();
    foreach (var job in jobs)
    {
        Console.WriteLine("\nJob: {0}", job.JobDetails);
        foreach (var bid in job.Bids)
        {
            Console.WriteLine("\tBid: {0} from {1}",
                              bid.Amount.ToString("C"), bid.Bidder);
        }
    }
}
```

The following is the output of the code in Listing 3-8:

```
Job: Re-surface Parking Log

        Bid: $948.00 from ABC Paving

        Bid: $1,028.00 from TopCoat Paving



Job: Build Driveway

        Bid: $502.00 from Ace Concrete
```

How It Works

We start out by adding a couple of jobs and a few bids for the jobs. After adding them to the context, we use **SaveChanges()** to save them to the database.

The current release of Entity Framework does not directly support working with multiple result sets. To solve the problem, we read the data using the familiar SqlClient pattern. This pattern involves

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

creating a `SqlConnection`, creating a `SqlCommand`, setting the command text to the name of the stored procedure, and calling `ExecuteReader()` to get a data reader.

With a reader in hand, we use the `Translate()` method on the object context to materialize instances of the Job entity from the reader. This method takes a reader, the entity set name, and a merge option. The entity set name is required because an entity can live in multiple entity sets. Entity Framework needs to know which to use.

The merge option parameter is a little more interesting. Using `MergeOption.AppendOnly` causes the new instances to be added to the object context and tracked. We use this option because we want to use Entity Framework's entity span to automatically fix up the associations between jobs and bids. We simply add to the context all the jobs and all the bids. Through the magic of entity span, Entity Framework will automatically associate the bids to the right jobs. This saves us a ton of tedious code. Entity span is not really magic, but it is something that comes in very handy.

A simpler version of the `Translate()` method does not require a `MergeOption`. This version materializes objects that are disconnected from the object context. This is subtly different from objects that are not tracked in that the objects are created completely outside of the object context. If you were to use this simpler `Translate()` to read the jobs, you would not be able to later materialize bids into the object context because Entity Framework would not have any reference to the associated jobs. Those jobs are completely disconnected from the object context.

We used `ToList()` to force the evaluation of each query. This is required because the `Translate()` method returns an `ObjectResult<T>`. It does not actually cause the results to be read from the reader. We need to force the results to be read from the reader before we can use `NextResult()` to advance to the next result set.

Although we didn't run into it in this example, it is important to note that `Translate()` bypasses the mapping layer of the model. If you try to map an inheritance hierarchy or use an entity that has complex type properties, `Translate()` will fail. `Translate()` requires that the `DbDataReader` have columns that match each property on the entity. This matching is done using simple name matching. If a column name can't be matched to a property, `Translate()` will fail.

3-8. Comparing Against a List of Values

Problem

You want to return entities whose property value matches one of the values in a given list.

Solution

Suppose you have a model like the one in Figure 3-9.

Figure 3-9. *A model for books and their categories*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

You want to find all the books in a given list of categories. To do this using LINQ or Entity SQL, follow the pattern in Listing 3-9.

Listing 3-9. Finding books in a list of categories using both LINQ and Entity SQL

```
using (var context = new EFRecipesEntities())
{
    var cat1 = new Category { Name = "Programming" };
    var cat2 = new Category { Name = "Databases" };
    var cat3 = new Category {Name = "Operating Systems"};
    context.Books.AddObject(new Book { Title = "F# In Practice", Category = cat1 });
    context.Books.AddObject(new Book { Title = "The Joy of SQL", Category = cat2 });
    context.Books.AddObject(new Book { Title = "Windows 7: The Untold Story",
                                       Category = cat3 });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Books (using LINQ)");
    List<string> cats = new List<string> { "Programming", "Databases" };
    var books = from b in context.Books
                where cats.Contains(b.Category.Name)
                select b;
    foreach (var book in books)
    {
        Console.WriteLine("'{0}' is in category: {1}", book.Title,
                          book.Category.Name);
    }
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Books (using ESQL)");
    var esql = @"select value b from Books as b
                 where b.Category.Name in {'Programming','Databases'}";
    var books = context.CreateQuery<Book>(esql);
    foreach (var book in books)
    {
        Console.WriteLine("'{0}' is in category: {1}", book.Title,
                          book.Category.Name);
    }
}
```

The following is the output of the code in Listing 3-9:

```
Books (using LINQ)

'F# In Practice' is in category: Programming
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
'The Joy of SQL' is in category: Databases

Books (using ESQL)

'F# In Practice' is in category: Programming

'The Joy of SQL' is in category: Databases
```

How It Works

For the LINQ query, we build a simple list of category names, use this list in a **contains** clause in the query. Entity Framework translates this to a SQL statement with an **in** clause, as shown in Listing 3-10.

Listing 3-10. The SQL statement created for the LINQ expression in Listing 3-9

```
SELECT
[Extent1].[BookId] AS [BookId],
[Extent1].[Title] AS [Title],
[Extent1].[CategoryId] AS [CategoryId]
FROM  [chapter3].[Books] AS [Extent1]
LEFT OUTER JOIN [chapter3].[Category] AS [Extent2] ON [Extent1].[CategoryId] =
[Extent2].[CategoryId]
WHERE [Extent2].[Name] IN (N'Programming',N'Databases')
```

It is interesting to note that the generated SQL statement in Listing 3-10 does not use parameters for the items in the **in** clause. This is different from the generated code we would see with LINQ to SQL where the items in the list would be parameterized. With this code, we don't run the risk of exceeding the parameters limit that is imposed by SQL Server.

If we are interested in finding all books in a given list of categories or books that are not yet categorized, we simply include null in the category list. The generated code is shown in Listing 3-11.

Listing 3-11. The SQL statement created for a LINQ expression like the one in Listing 3-9, but with a ***null*** *in the list of categories*

```
SELECT
[Extent1].[BookId] AS [BookId],
[Extent1].[Title] AS [Title],
[Extent1].[CategoryId] AS [CategoryId]
FROM  [chapter3].[Books] AS [Extent1]
LEFT OUTER JOIN [chapter3].[Category] AS [Extent2] ON [Extent1].[CategoryId] =
[Extent2].[CategoryId]
WHERE [Extent2].[Name] IN (N'Programming',N'Databases')
       OR [Extent2].[Name] IS NULL
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3-9. Building and Executing a Query Against an ObjectSet<T>

Problem

You want to build and execute a query against an **ObjectSet<T>**.

Solution

Let's suppose you have model like the one in Figure 3-10 and you want to build and execute a query against the **ObjectSet<Patient>**.

Figure 3-10. *A model with a Patient entity type*

For each of the entities we create in our model, Entity Framework creates code for the object context that contains a definition of a property of type **ObjectSet<T>** where T is our entity type. In our case, the object context contains a property called Patients that is of type **ObjectSet<Patient>**. To query against this, follow one of the patterns in Listing 3-12.

Listing 3-12. Using three slightly different approaches to build and execute a query against an ***ObjectSet<T>***

```
using (var context = new EFRecipesEntities())
{
    context.Patients.AddObject(new Patient { Name = "Jill Stevens",
                                             City = "Dallas" });
    context.Patients.AddObject(new Patient { Name = "Bill Azle",
                                             City = "Fort Worth" });
    context.Patients.AddObject(new Patient { Name = "Karen Stanford",
                                             City = "Raytown" });
    context.Patients.AddObject(new Patient { Name = "David Frazier",
                                             City = "Dallas" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Using LINQ Builder Methods");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    var patients = context.Patients.Where(p => p.City == "Dallas");
    foreach (var patient in patients)
    {
        Console.WriteLine("{0} is in {1}", patient.Name, patient.City);
    }
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("\nUsing Entity SQL");
    var patients = context.CreateQuery<Patient>(
              @"select value p from Patients as p where p.City = 'Dallas'");
    foreach (var patient in patients)
    {
        Console.WriteLine("{0} is in {1}", patient.Name, patient.City);
    }
}
using (var context = new EFRecipesEntities())
{
    Console.WriteLine("\nUsing ESQL Builder Methods");
    var patients = context.CreateObjectSet<Patient>("Patients")
                          .Where("it.City = 'Dallas'");
    foreach (var patient in patients)
    {
        Console.WriteLine("{0} is in {1}", patient.Name, patient.City);
    }
}
```

The following is the output of the code in Listing 3-12:

```
Using LINQ Builder Methods

Jill Stevens is in Dallas

David Frazier is in Dallas


Using Entity SQL

Jill Stevens is in Dallas

David Frazier is in Dallas


Using ESQL Builder Methods

Jill Stevens is in Dallas

David Frazier is in Dallas
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Each of our entities in a model is exposed as an `ObjectSet<T>`, which has everything that `ObjectQuery<T>` has plus a few methods like `AddObject()`, `Attach()`, and `DeleteObject()`. When we build a query against `ObjectSet<T>`, we get an `ObjectQuery<T>`. Of course, we can continue to compose queries on type of `ObjectQuery<T>`.

In Listing 3-12, we demonstrate three common approaches to building a query. In the first approach, we use the `Where()` method and a lambda expression to filter the collection to patients in Dallas.

In the second approach, we use the `CreateQuery()` method and an Entity SQL expression to get all the patients in Dallas.

In the last approach, we use the `CreateObjectSet<Patient>()` method and the `Where()` method with an Entity SQL expression to filter the collection.

Although we didn't show it in these examples, `ObjectSet<T>` has a `MergeOption` property that defines how the materialized instances of our entities are to be loaded, tracked, and merged in the object context. Table 3-1 summarizes these merge options.

Table 3-1. *Merge Options Available on* `ObjectSet<T>`

Merge Option	Description
AppendOnly	Default behavior; add new instances to the object context
OverwriteChanges	Overwrite any changes made to objects in the object context
PreserveChanges	Just reset the original values; don't overwrite changes in the object context
NoTracking	Don't track objects in the object context

3-10. Returning a Primitive Type From a Query

Problem

You want to return only a particular property of an entity type from a query.

Solution

Let's say we have an Organization entity type as shown in the model in Figure 3-11.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 3-11. *A model with an Organization entity type*

Suppose that you have a query that you use in several places in your application. In this particular use, you don't need the entire entity; you just need one property, say the city, from the entity. For simplicity, let's say your query returns all the organizations in the state of Texas and you want to get just the cities from the query.

To get just the cities, follow the pattern in Listing 3-13.

Listing 3-13. Retrieving a primitive type using both LINQ and Entity SQL

```
using (var context = new EFRecipesEntities())
{
    var o1 = new Organization { Name = "ABC Electric", City = "Azle",
                                State = "TX" };
    var o2 = new Organization { Name = "PowWow Pests", City = "Miami",
                                State = "FL" };
    var o3 = new Organization { Name = "Grover Grass & Seed",
                                City = "Fort Worth", State = "TX" };
    context.Organizations.AddObject(o1);
    context.Organizations.AddObject(o2);
    context.Organizations.AddObject(o3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var query = context.Organizations.Where("it.State = 'TX'");
    Console.WriteLine("Cities (using LINQ)");
    var cities = query.Select(o => o.City).Distinct().OrderBy(c => c);
    foreach (var city in cities)
    {
        Console.WriteLine("{0}", city);
    }

    Console.WriteLine("Cities (using eSQL)");
    cities = query.SelectValue<string>("distinct it.City").OrderBy("it");
    foreach (var city in cities)
    {
        Console.WriteLine("{0}", city);
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following is the output of the code in Listing 3-13:

```
Cities (using LINQ)
Azle
Fort Worth
Cities (using eSQL)
Azle
Fort Worth
```

How It Works

We start in Listing 3-13 by inserting a few organizations. Once these are in place, we create a new object context and build our base query. This query simply retrieves all the organizations in the state of Texas.

Next, using LINQ, we use the **Select()** method to project just the city property from the result set. This set may contain duplicate cities, so we apply the **Distinct()** method to eliminate these duplicates. Finally, just for good measure, we sort the cities.

For Entity SQL, we use the **SelectValue()** method passing in an Entity SQL expression that uses the distinct operator to remove duplicates. We sort the results.

In both cases, we take a base query and we use builder methods to compose the additional operations of projection (using **Select()** for LINQ and **SelectValue()** for Entity SQL) and ordering to get the final collection of city names.

3-11. Filtering Related Entities

Problem

You want to want to retrieve some, but not all, of the related entities.

Solution

Let's say you have a model like the one in Figure 3-12.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

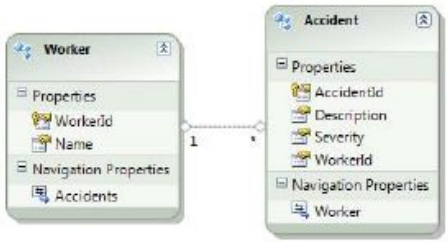


Figure 3-12. *A model for a Worker and their Accidents*

In this model, we have a Worker who has experienced zero or more accidents. Each accident is classified by its severity. We want to retrieve all workers, but we are interested only in serious accidents. These are accidents with a severity greater than 2.

To retrieve all the workers, but to limit the accidents retrieved to just the serious ones, follow the pattern in Listing 3-14.

Listing 3-14. Retrieving serious accidents using anonymous types and using ***CreateSourceQuery()***

```
using (var context = new EFRecipesEntities())
{
    var worker1 = new Worker { Name = "John Kearney" };
    var worker2 = new Worker { Name = "Nancy Roberts" };
    var worker3 = new Worker { Name = "Karla Gibbons" };
    context.Accidents.AddObject(new Accident {
                  Description = "Cuts and contusions",
                  Severity = 3, Worker = worker1 });
    context.Accidents.AddObject(new Accident {
                  Description = "Broken foot",
                  Severity = 4, Worker = worker1});
    context.Accidents.AddObject(new Accident {
                  Description = "Fall, no injuries",
                  Severity = 1, Worker = worker2});
    context.Accidents.AddObject(new Accident {
                  Description = "Minor burn",
                  Severity = 3, Worker = worker2});
    context.Accidents.AddObject(new Accident {
                  Description = "Back strain",
                  Severity = 2, Worker = worker3});
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = false;
    var query = from w in context.Workers
                select new
                {
                    Worker = w,
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                    Accidents = w.Accidents.Where(a => a.Severity > 2)
                };
    query.ToList();
    var workers = query.Select(r => r.Worker);
    Console.WriteLine("Workers with serious accidents...");
    foreach (var worker in workers)
    {
        Console.WriteLine("{0} had the following accidents", worker.Name);
        if (worker.Accidents.Count == 0)
            Console.WriteLine("\t--None--");
        foreach (var accident in worker.Accidents)
        {
            Console.WriteLine("\t{0}, severity: {1}",
                  accident.Description, accident.Severity.ToString());
        }
    }
}

Console.WriteLine();

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = false;
    foreach (var worker in context.Workers)
    {
        Console.WriteLine("{0} had the following accidents", worker.Name);
        var accidents = worker.Accidents.CreateSourceQuery()
                        .Where(a => a.Severity > 2);
        worker.Accidents.Attach(accidents);
        if (worker.Accidents.Count == 0)
            Console.WriteLine("\t--None--");
        foreach (var accident in accidents)
        {
            Console.WriteLine("\t{0}, severity: {1}",
                accident.Description, accident.Severity.ToString());
        }
    }
}
```

The following is the output of the code in Listing 3-14:

```
Workers with serious accidents...

John Kearney had the following accidents

        Cuts and contusions, severity: 3

        Broken foot, severity: 4
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Nancy Roberts had the following accidents
        Minor burn, severity: 3
Karla Gibbons had the following accidents
        --None--

John Kearney had the following accidents
        Cuts and contusions, severity: 3
        Broken foot, severity: 4
Nancy Roberts had the following accidents
        Minor burn, severity: 3
Karla Gibbons had the following accidents
        --None--
```

How It Works

As you will see in Chapter 5, when we want to eagerly load a related collection, we often use the **Include()** method with a query path. However, the **Include()** method does not allow filtering on the related entities. In this recipe, we show two slightly different ways to load and filter related entities.

In the first block of code, we create a few workers and assign them accidents of varying levels of severity. Granted, it's a little creepy to assign accidents to people, but it's all in the name of getting some data to work with.

In the first approach, we select from all the workers and project the results into an anonymous type. The type includes the worker and the collection of accidents. For the accidents, we filter the collection to get just the serious accidents.

The very next line is important. Here we force the evaluation of the query by calling the **ToList()** method. This brings all the workers and all the serious accidents into the Object Context. The anonymous type didn't attach the accidents to the workers, but by bringing them into the Object Context, Entity Framework will fix up the navigation properties, attaching each collection of serious accidents to the appropriate worker. This process, commonly known as Entity Span, is a powerful yet subtle side effect that happens behind the scenes to fix up relationships between entities as they are materialized in the Object Context.

We've turned off lazy loading (we'll talk more about lazy loading in Chapter 5) so that only the accidents in our filter are loaded. With lazy loading on, all the accidents would get loaded when we referenced each worker's accidents. That would defeat our filter.

Once we have the collection, we iterate through it, printing out each worker and their serious accidents. If a worker didn't have any serious accidents, we print none to indicate their stellar safety record.

For the second approach, we use the **CreateSourceQuery()** method to append our filter onto the query that Entity Framework will use to gather up all the accidents for the worker. We have more to say about **CreateSourceQuery()** in Chapter 5. Once we have the collection of serious accidents for the

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

worker, we **attach()** it to the worker. After attaching the collection, we iterate through the collection printing out the serious accidents.

3-12. Applying a Left Outer Join

Problem

You want to combine the properties of two entities using a left outer join.

Solution

Licensed by Matt Allen 2175280

Suppose you have a model like the one in Figure 3-13.

Figure 3-13. *Our model with a Product entity type and its related TopSelling entity type*

The top-selling products have a related TopSelling entity. Of course, not all products are top sellers, and that's why the relationship is one to zero or one. When a product is a top seller, the related TopSeller entity also contains the customer rating for the product. You want to find all the products and their related TopSeller entities even if, in some cases, the product is not a top seller. In database terms, this is called a left outer join.

The code in Listing 3-15 demonstrates three slightly different approaches to this problem.

Listing 3-15. Doing a left outer join between entities

```
using (var context = new EFRecipesEntities())
{
    var p1 = new Product { Name = "Trailrunner Backpack" };
    var p2 = new Product { Name = "Green River Tent",
                           TopSelling = new TopSelling { Rating = 3 } };
    var p3 = new Product { Name = "Prairie Home Dutch Oven",
                           TopSelling = new TopSelling { Rating = 4 } };
    var p4 = new Product { Name = "QuickFire Fire Starter",
                           TopSelling = new TopSelling { Rating = 2 } };
    context.Products.AddObject(p1);
    context.Products.AddObject(p2);
    context.Products.AddObject(p3);
    context.Products.AddObject(p4);
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    var products = from p in context.Products
                   orderby p.TopSelling.Rating descending
                   select p;
    Console.WriteLine("Top selling products sorted by rating");
    foreach (var product in products)
    {
        if (product.TopSelling != null)
            Console.WriteLine("\t{0} [rating: {1}]", product.Name,
                              product.TopSelling.Rating.ToString());
    }
}

using (var context = new EFRecipesEntities())
{
    var products = from p in context.Products
                   join t in context.TopSellings on
                      p.ProductId equals t.ProductId into g
                   from tps in g.DefaultIfEmpty()
                   orderby tps.Rating descending
                   select new
                       {
                           Name = p.Name,
                           Rating = tps.Rating == null ? 0 : tps.Rating
                       };

    Console.WriteLine("\nTop selling products sorted by rating");
    foreach (var product in products)
    {
        if (product.Rating != 0)
            Console.WriteLine("\t{0} [rating: {1}]", product.Name,
                              product.Rating.ToString());
    }
}

using (var context = new EFRecipesEntities())
{
    var esql = @"select value p from products as p
                 order by case when p.TopSelling is null then 0
                               else p.TopSelling.Rating end desc";
    var products = context.CreateQuery<Product>(esql);
    Console.WriteLine("\nTop selling products sorted by rating");
    foreach (var product in products)
    {
        if (product.TopSelling != null)
            Console.WriteLine("\t{0} [rating: {1}]", product.Name,
                              product.TopSelling.Rating.ToString());
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following is the output of the code in Listing 3-15:

```
Top selling products sorted by rating

        Prairie Home Dutch Oven [rating: 4]

        Green River Tent [rating: 3]

        QuickFire Fire Starter [rating: 2]


Top selling products sorted by rating

        Prairie Home Dutch Oven [rating: 4]

        Green River Tent [rating: 3]

        QuickFire Fire Starter [rating: 2]


Top selling products sorted by rating

        Prairie Home Dutch Oven [rating: 4]

        Green River Tent [rating: 3]

        QuickFire Fire Starter [rating: 2]
```

How It Works

In Listing 3-15, we show three slightly different solutions. The first solution is the simplest because Entity Framework handles the join automatically for related entities. The entities are in a one to zero or one association. When the product entities are materialized, any associated top sellers are also materialized. The TopSeller navigation property is either set to the associated TopSeller entity or null if no TopSeller exists.

In some cases, you might not have a relationship between the entities that you want to join. In these cases, you can explicitly join the entities projecting the results into an anonymous type. We need to project into an anonymous type because the unrelated entities won't have navigation properties so we wouldn't otherwise be able to reference the related entity.

The code in the second query block illustrates this approach. Here we join the entities on the ProductId key and put the result into g. Now, from g we apply the **DefaultIfEmpty()** method to fill in nulls when g is empty. This gives us the left inner join. We include an **orderby** clause to order the results by the rating. Finally, we project the results into an anonymous type.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In the third solution, we show you how to do the left inner join more explicitly using Entity SQL.

3-13. Ordering by Derived Types

Problem

You are using Table per Hierarchy inheritance and you want to sort results by the derived type.

Solution

Let's suppose you have a model like the one in Figure 3-14.

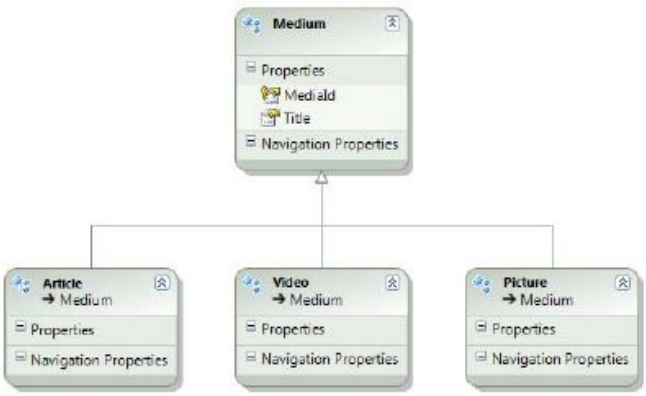


Figure 3-14. *A model using Table per Hierarchy inheritance with three derived types*

This model uses Table per Hierarchy inheritance. The Medium entity has a discriminator property whose values determine which derived type is represented by a row from the database. This discriminator column has a value of 1 for the Article type, a value of 2 for the Video type, and a value of 3 for the Picture type. Because the property is used only to determine the derived type, it is not shown as part of the entity.

You want to query the model for all media and sort the results by the derived types: Article, Video, and Picture. To do this, follow the pattern in Listing 3-16.

Listing 3-16. Sorting Table per Hierarchy inheritance by type

```
using (var context = new EFRecipesEntities())
{
    context.Media.AddObject(new Article {
                   Title = "Woodworkers' Favorite Tools" });
    context.Media.AddObject(new Article {
                   Title = "Building a Cigar Chair" });
    context.Media.AddObject(new Video {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                Title = "Upholstering the Cigar Chair" });
    context.Media.AddObject(new Video {
                Title = "Applying Finish to the Cigar Chair" });
    context.Media.AddObject(new Picture {
                Title = "Photos of My Cigar Chair" });
    context.Media.AddObject(new Video {
                Title = "Tour of My Woodworking Shop" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var allMedia = from m in context.Media
                   let mediatype = m is Article ? 1 :
                                   m is Video ? 2 : 3
                   orderby mediatype
                   select m;
    Console.WriteLine("All Media sorted by type...");
    foreach (var media in allMedia)
    {
        Console.WriteLine("Title: {0} [{1}]", media.Title, media.GetType().Name);
    }
}
```

The following is the output of the code in Listing 3-16:

```
All Media sorted by type...

Title: Woodworkers' Favorite Tools [Article]

Title: Building a Cigar Chair [Article]

Title: Upholstering the Cigar Chair [Video]

Title: Applying Finish to the Cigar Chair [Video]

Title: Tour of My Woodworking Shop [Video]

Title: Photos of My Cigar Chair [Picture]
```

How It Works

When we use Table per Hierarchy inheritance we leverage a column in the table to distinguish which derived type represents any given row. This column, often referred to as the discriminator column, can't be mapped to a property of the base entity. Because we don't have a property with the discriminator value, we need to create a variable to hold comparable discriminator values so that we can do the sort. To do this, we use a **let** clause creating the mediatype variable. We use a conditional statement to assign an integer to this variable based on the type of the media. For Articles we assign the value 1. For Videos,

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

we assign the value 2. We assign 3 to anything else, which will always be of type Picture because we don't have any other derived types left.

3-14. Paging and Filtering

Problem

You want to create query with a filter and paging.

Solution

Let's say you have a Customer entity type in a model, as shown in Figure 3-15.

Figure 3-15. *A model with a Customer entity type*

You have an application that displays customers based on a filter. Your company has many customers (perhaps millions!) and to keep the user experience as responsive as possible, you want to show only a limited number of customers on each page. To create a query that both filters the customers and returns a manageable set for each results page in your application, follow the pattern in Listing 3-17.

Listing 3-17. Filtering and paging a query

```
using (var context = new EFRecipesEntities())
{
    context.Customers.AddObject(new Customer { Name = "Roberts, Jill",
                                Email = "jroberts@abc.com" });
    context.Customers.AddObject(new Customer { Name = "Robertson, Alice",
                                Email = "arob@gmail.com" });
    context.Customers.AddObject(new Customer { Name = "Rogers, Steven",
                                Email = "srogers@termite.com" });
    context.Customers.AddObject(new Customer { Name = "Roe, Allen",
                                Email = "allenr@umc.com" });
    context.Customers.AddObject(new Customer { Name = "Jones, Chris",
                                Email = "cjones@ibp.com" });
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    string match = "Ro";
    int pageIndex = 0;
    int pageSize = 3;

    var customers = context.Customers.Where(c => c.Name.StartsWith(match))
                        .OrderBy(c => c.Name)
                        .Skip(pageIndex * pageSize)
                        .Take(pageSize);
    Console.WriteLine("Customers Ro*");
    foreach (var customer in customers)
    {
        Console.WriteLine("{0} [email: {1}]", customer.Name, customer.Email);
    }
}
using (var context = new EFRecipesEntities())
{
    string match = "Ro%";
    int pageIndex = 0;
    int pageSize = 3;

    var customers = context.Customers.Where("it.Name like @Name",
                                  new ObjectParameter("Name", match))
                            .Skip("it.Name", "@Skip",
                                  new ObjectParameter("Skip", pageIndex))
                            .Top("@Limit",
                                  new ObjectParameter("Limit", pageSize));
    Console.WriteLine("\nCustomers Ro*");
    foreach (var customer in customers)
    {
        Console.WriteLine("{0} [email: {1}]", customer.Name, customer.Email);
    }
}

using (var context = new EFRecipesEntities())
{
    string match = "Ro%";
    int pageIndex = 0;
    int pageSize = 3;

    var esql = @"select value c from Customers as c
                 where c.Name like @Name
                 order by c.Name
                 skip @Skip limit @Limit";
    Console.WriteLine("\nCustomers Ro*");
    var customers = context.CreateQuery<Customer>(esql, new[]
                      {
                        new ObjectParameter("Name",match),
                        new ObjectParameter("Skip",pageIndex * pageSize),
                        new ObjectParameter("Limit",pageSize)
                      });
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var customer in customers)
    {
        Console.WriteLine("{0} [email: {1}]", customer.Name, customer.Email);
    }
}
```

The following is the output from the code in Listing 3-17:

```
Customers Ro*

Roberts, Jill [email: jroberts@abc.com]

Robertson, Alice [email: arob@gmail.com]

Roe, Allen [email: allenr@umc.com]


Customers Ro*

Roberts, Jill [email: jroberts@abc.com]

Robertson, Alice [email: arob@gmail.com]

Roe, Allen [email: allenr@umc.com]


Customers Ro*

Roberts, Jill [email: jroberts@abc.com]

Robertson, Alice [email: arob@gmail.com]

Roe, Allen [email: allenr@umc.com]
```

How It Works

In Listing 3-17 we show three different solutions to the problem. In the first solution, we use LINQ builder methods to construct the query. We use the **Where()** method to filter the results to customers whose last name starts with Ro. Because we are using the **StartsWith()** method inside the lambda expression, we don't need to use a SQL wildcard expression such as "Ro%".

After filtering, we use the **OrderBy()** method to order the results. Ordered results are required by the **Skip()** method. We use the **Skip()** method to move over **pageIndex** number of pages, each of size **pageSize**. We limit the results with the **Take()** method. We only need to take one page of results.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In the next solution, we use Entity SQL builder methods to construct the query. We use the **Where()** and **Skip()** builder methods as we did in the LINQ solution but this time with Entity SQL syntax. For limiting the result set size, we use the **Top()** method. One difference is that we don't need to use the **OrderBy()** method. The **Skip()** method takes a parameter to name the column on which to perform the ordering. Ordering is important for the **Skip()** method because without it, the query results would not be repeatable.

For the last solution we construct a complete, parameterized Entity SQL expression. This is perhaps the most familiar way to solve the problem, but it exposes some of the inherent mismatch between a query language expressed in a string and executable code expressed, in this case, in C#.

3-15. Grouping by Date

Problem

You have an entity type with a DateTime property and you want to group instances of this type based on just the date portion of the property.

Solution

Let's say you have a Registration entity type in your model and the Registration type has a DateTime property. Your model might look like the one in Figure 3-16.

Figure 3-16. *A model with a single Registration entity type. The entity type's RegistrationDate property is a DateTime.*

We want to group all the registrations by just the date portion of the RegistrationDate property. You might be tempted in LINQ to group by **RegistrationDate.Date**. Although this will compile, you will receive a runtime error complaining that Date can't be translated into SQL. To group by just the date portion of the RegistrationDate, follow the pattern in Listing 3-18.

Listing 3-18. Grouping by the date portion of a DateTime property

```
using (var context = new EFRecipesEntities())
{
    context.Registrations.AddObject(new Registration {
          StudentName = "Jill Rogers",
          RegistrationDate = DateTime.Parse("12/03/2009 9:30 pm") });
    context.Registrations.AddObject(new Registration {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            StudentName = "Steven Combs",
            RegistrationDate = DateTime.Parse("12/03/2009 10:45 am") });
    context.Registrations.AddObject(new Registration {
            StudentName = "Robin Rosen",
            RegistrationDate = DateTime.Parse("12/04/2009 11:18 am") });
    context.Registrations.AddObject(new Registration {
            StudentName = "Allen Smith",
            RegistrationDate = DateTime.Parse("12/04/2009 3:31 pm") });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var groups = from r in context.Registrations
                 group r by EntityFunctions.TruncateTime(r.RegistrationDate)
                    into g
                 select g;
    foreach (var element in groups)
    {
        Console.WriteLine("Registrations for {0}",
               ((DateTime)element.Key).ToShortDateString());
        foreach (var registration in element)
        {
            Console.WriteLine("\t{0}", registration.StudentName);
        }
    }
}
```

The following is the output of the code in Listing 3-18:

```
Registrations for 12/3/2009

        Jill Rogers

        Steven Combs

Registrations for 12/4/2009

        Robin Rosen

        Allen Smith
```

How It Works

The key to grouping the registrations by the date portion of the RegistrationDate property is to use the **Truncate()** function. This built-in Entity Framework function extracts just the date portion of the DateTime value. We'll have a lot more to say about functions in Chapter 11.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3-16. Flattening Query Results

Problem

You have two entity types in a one-to-many association and you want, in one query, to obtain a flatten projection of all the entities in the association.

Solution

Let's say you have a couple of entity types in a one-to-many association. Perhaps your model looks something like the one in Figure 3-17.

Licensed by Matt Allen 2175280

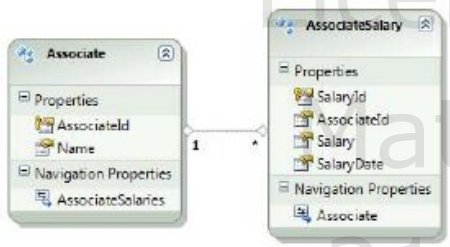


Figure 3-17. *A model with an Associate entity type representing an associate, and an AssociateSalary entity type representing the salary history for the associate*

You want to get all the associates and all their salary history in one query. There may be some new hires that are in the system but don't yet have a salary set. You want your query results to include these associates as well.

To query the model and get the results you want, follow the pattern in Listing 3-19.

Listing 3-19. Flattening out the results using both LINQ and Entity SQL

```
using (var context = new EFRecipesEntities())
{
    var assoc1 = new Associate { Name = "Janis Roberts" };
    var assoc2 = new Associate { Name = "Kevin Hodges" };
    var assoc3 = new Associate { Name = "Bill Jordan" };
    var salary1 = new AssociateSalary { Salary = 39500M,
                        SalaryDate = DateTime.Parse("8/14/09") };
    var salary2 = new AssociateSalary { Salary = 41900M,
                        SalaryDate = DateTime.Parse("2/5/10") };
    var salary3 = new AssociateSalary { Salary = 33500M,
                        SalaryDate = DateTime.Parse("10/08/09") };
    assoc2.AssociateSalaries.Add(salary1);
    assoc2.AssociateSalaries.Add(salary2);
    assoc3.AssociateSalaries.Add(salary3);
    context.Associates.AddObject(assoc1);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        context.Associates.AddObject(assoc2);
        context.Associates.AddObject(assoc3);
        context.SaveChanges();
    }

    using (var context = new EFRecipesEntities())
    {
        Console.WriteLine("Using LINQ...");
        var allHistory = from a in context.Associates
                         from ah in a.AssociateSalaries.DefaultIfEmpty()
                         orderby a.Name
                         select new
                         {
                             Name = a.Name,
                             Salary = (decimal ?) ah.Salary,
                             Date = (DateTime ?) ah.SalaryDate
                         };
        Console.WriteLine("Associate Salary History");
        foreach (var history in allHistory)
        {
            if (history.Salary.HasValue)
                Console.WriteLine("{0} Salary on {1} was {2}", history.Name,
                                  history.Date.Value.ToShortDateString(),
                                  history.Salary.Value.ToString("C"));
            else
                Console.WriteLine("{0} --", history.Name);
        }
    }

    using (var context = new EFRecipesEntities())
    {
        Console.WriteLine("\nUsing Entity SQL...");
        var esql = @"select a.Name, h.Salary, h.SalaryDate
                     from Associates as a outer apply
                       a.AssociateSalaries as h order by a.Name";
        var allHistory = context.CreateQuery<DbDataRecord>(esql);
        foreach (var history in allHistory)
        {
            if (history["Salary"] != DBNull.Value)
                Console.WriteLine("{0} Salary on {1:d} was {2:c}", history["Name"],
                                  history["SalaryDate"], history["Salary"]);
            else
                Console.WriteLine("{0} --",history["Name"]);
        }
    }
```

The following is the output of the code in Listing 3-19:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Using LINQ...

Associate Salary History

Bill Jordan Salary on 10/8/2009 was $33,500.00

Janis Roberts --

Kevin Hodges Salary on 8/14/2009 was $39,500.00

Kevin Hodges Salary on 2/5/2010 was $41,900.00

Using Entity SQL...

Bill Jordan Salary on 10/8/2009 was $33,500.00

Janis Roberts --

Kevin Hodges Salary on 8/14/2009 was $39,500.00

Kevin Hodges Salary on 2/5/2010 was $41,900.00
```

How It Works

To flatten the query results we followed the strategy in Recipe 12 in this chapter and used a nested **from** clause and the **DefaultIfEmpty()** method to get a left outer join between the tables. The **DefaultIfEmpty()** method ensured that we have rows from the left side (the Associate entities), even if there are no corresponding rows on the right side (AssociateSalary entities). We project the results into an anonymous type being careful to capture null values for the salary and salary date when there are no corresponding AssociateSalary entities.

For the Entity SQL solution, we use the **outer apply** operator to create unique pairings between each Associate entity and AssociateSalary entity. Both the **cross** and **outer apply** operators were introduced in SQL Server 2005.

3-17. Grouping by Multiple Properties

Problem

You want to group the results of a query by multiple properties.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have a model with an Event entity type like the one in Figure 3-18. Event has a name, city, and state. You want to group events by state and then by city.

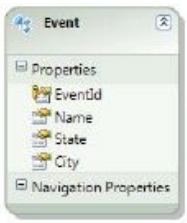


Figure 3-18. *A model with an Event entity type which as properties for the event's name, state, and city*

To get all the events grouped by state and then city, follow the pattern in Listing 3-20.

Listing 3-20.

```
using (var context = new EFRecipesEntities())
{
    context.Events.AddObject(new Event { Name = "TechFest 2010",
                                         State = "TX", City = "Dallas" });
    context.Events.AddObject(new Event { Name = "Little Blue River Festival",
                                         State = "MO", City = "Raytown" });
    context.Events.AddObject(new Event { Name = "Fourth of July Fireworks",
                                         State = "MO", City = "Raytown" });
    context.Events.AddObject(new Event { Name = "BBQ Ribs Championship",
                                         State = "TX", City = "Dallas" });
    context.Events.AddObject(new Event { Name = "Thunder on the Ohio",
                                         State = "KY", City = "Louisville" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Using LINQ");
    var results = from e in context.Events
                  group e by new { e.State, e.City } into g
                  select new
                      {
                          State = g.Key.State,
                          City = g.Key.City,
                          Events = g                    };
    Console.WriteLine("Events by State and City...");
    foreach (var item in results)
    {
        Console.WriteLine("{0}, {1}", item.City, item.State);
        foreach (var ev in item.Events)
        {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            Console.WriteLine("\t{0}", ev.Name);
        }
    }
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("\nUsing Entity SQL");
    var esql = @"select e.State, e.City, GroupPartition(e) as Events
                 from Events as e
                 group by e.State, e.City";
    var records = context.CreateQuery<DbDataRecord>(esql);
    Console.WriteLine("Events by State and City...");
    foreach (var rec in records)
    {
        Console.WriteLine("{0}, {1}", rec["City"], rec["State"]);
        var events = (List<Event>)rec["Events"];
        foreach (var ev in events)
        {
            Console.WriteLine("\t{0}", ev.Name);
        }
    }
}
```

The following is the output of the code in Listing 3-20:

```
Using LINQ

Events by State and City...

Louisville, KY

        Thunder on the Ohio

Raytown, MO

        Little Blue River Festival

        Fourth of July Fireworks

Dallas, TX

        TechFest 2010

        BBQ Ribs Championship
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Using Entity SQL

Events by State and City...

Louisville, KY

        Thunder on the Ohio

Raytown, MO

        Little Blue River Festival

        Fourth of July Fireworks

Dallas, TX

        TechFest 2010

        BBQ Ribs Championship
```

How It Works

In Listing 3-20, we show two different solutions. The first solution uses LINQ and the **group by** operator to group the results by state and city. When using the **group by** operator for multiple properties, we create an anonymous type for the grouping. We use an **into** clause to send the groups to **g**.

We project the results from **g** into a new anonymous type getting the State from the group key's State field (from the first anonymous type) and the City from the group key's City field. For the events, we simply select all the members of the group.

For the Entity SQL approach, we can only project columns used in the **group by** clause, a constant value, or a computed value from using an aggregate function. In our case, we project the state, city, and the collection of events for each grouping.

3-18. Using Bitwise Operators in a Filter

Problem

You want to use bitwise operators to filter a query.

Solution

Let's say you have an entity type with an integer property that you want to use as a set of bit flags. You'll use some of the bits in this property to represent the presence or absence of some particular attribute for the entity. For example, suppose you have an entity type for patrons of a local art gallery. Some patrons

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

contribute money. Some volunteer during gallery hours. A few patrons serve on the board of directors. A few patrons support the art gallery in more than one way. A model with this entity type is shown in Figure 3-19.

Figure 3-19. *A Patron entity type with a SponsorType property that we use as a collection of bit flags indicating the sponsorship type for the patron*

We want to query for patrons and filter on the type of sponsorship provided by the patron. To do this, follow the pattern in Listing 3-21.

Listing 3-21. Using bitwise operators in a query

```
static void Main(string[] args)
{
    RunExample();
}

public enum SponsorTypes
{
    ContributesMoney = 1,
    Volunteers = 2,
    IsABoardMember = 4
};

static void RunExample()
{
    using (var context = new EFRecipesEntities())
    {
        context.Patrons.AddObject(new Patron { Name = "Jill Roberts",
                   SponsorType = (int)SponsorTypes.ContributesMoney });
        context.Patrons.AddObject(new Patron { Name = "Ryan Keyes",
                   SponsorType = (int)(SponsorTypes.ContributesMoney |
                                       SponsorTypes.IsABoardMember)});
        context.Patrons.AddObject(new Patron {Name = "Karen Rosen",
                   SponsorType = (int)SponsorTypes.Volunteers});
        context.Patrons.AddObject(new Patron {Name = "Steven King",
                   SponsorType = (int)(SponsorTypes.ContributesMoney |
                                       SponsorTypes.Volunteers)});
        context.SaveChanges();
    }

    using (var context = new EFRecipesEntities())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    {
        Console.WriteLine("Using LINQ...");
        var sponsors = from p in context.Patrons
                       where (p.SponsorType &
                              (int)SponsorTypes.ContributesMoney) != 0
                       select p;
        Console.WriteLine("Patrons who contribute money");
        foreach (var sponsor in sponsors)
        {
            Console.WriteLine("\t{0}", sponsor.Name);
        }
    }

    using (var context = new EFRecipesEntities())
    {
        Console.WriteLine("\nUsing Entity SQL...");
        var esql = @"select value p from Patrons as p
                     where BitWiseAnd(p.SponsorType, @type) <> 0";
        var sponsors = context.CreateQuery<Patron>(esql,
           new ObjectParameter("type", (int)SponsorTypes.ContributesMoney));
        Console.WriteLine("Patrons who contribute money");
        foreach (var sponsor in sponsors)
        {
            Console.WriteLine("\t{0}", sponsor.Name);
        }
    }
}
```

The following is the output of the code in Listing 3-21:

```
Using LINQ...

Patrons who contribute money

        Jill Roberts

        Ryan Keyes

        Steven King



Using Entity SQL...

Patrons who contribute money

        Jill Roberts
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Ryan Keyes

Steven King
```

How It Works

In our model, the Patron entity type packs multiple bit flags into a single integer property. A patron can sponsor the gallery in a number of ways. Each type of sponsorship is represented in a different bit in the SponsorType property. We represented each of the ways a sponsor can contribute in the `SponsorTypes enum`. We were careful to assign integers in power of 2 increments for each sponsor type. This means that each will have exactly one unique bit in the bits of the SponsorType property.

When we inserted a few patrons, we assign the sponsorship type to the SponsorType property. For patrons that contribute in more than one way, we simply use the bitwise OR (|) operator to build the bit pattern representing all the ways the patron contributes to the gallery.

For the LINQ query, we use the bitwise `AND (&)` operator to extract the bit for the `ContributesMoney` flag from the SponsorType property value. If the result is non-zero, then the patron has the `ContributesMoney` flag set. If we needed to find patrons that contribute in more than one way, we would OR all the SponsorTypes we're interested in together before we used the `AND` operator to extract one or more set bits.

The second solution demonstrates the same approach using Entity SQL. Here we use the `BitWiseAnd()` function to extract the set bit. Entity SQL supports a full complement of bitwise functions.

3-19. Joining on Multiple Columns

Problem

You want to join two entity types on multiple properties.

Solution

Let's say you have the model like the one in Figure 3-20. The Account entity type is in a one-to-many association with the Order type. Each account may have many orders while each order is associated with exactly one order. You want to find all the orders that are being shipped to a same city and state as the account.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

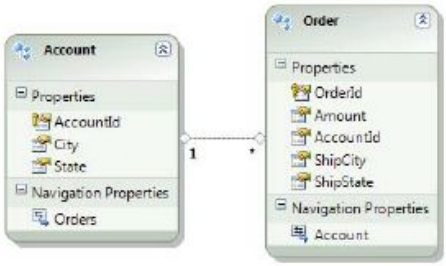


Figure 3-20. *A model with an Account entity type and its associated Order entity type*

To find the orders, follow the pattern in Listing 3-22.

Listing 3-22. Using a join on multiple properties to find all the orders being shipped to the account's city and state

```
using (var context = new EFRecipesEntities())
{
    var a1 = new Account { City = "Raytown", State = "MO" };
    a1.Orders.Add(new Order { Amount = 223.09M, ShipCity = "Raytown",
                              ShipState = "MO" });
    a1.Orders.Add(new Order { Amount = 189.32M, ShipCity = "Olathe",
                              ShipState = "KS" });

    var a2 = new Account { City = "Kansas City", State = "MO" };
    a2.Orders.Add(new Order { Amount = 99.29M, ShipCity = "Kansas City",
                              ShipState = "MO" });

    var a3 = new Account { City = "North Kansas City", State = "MO"};
    a3.Orders.Add(new Order { Amount = 102.29M, ShipCity = "Overland Park",
                              ShipState = "KS" });
    context.Accounts.AddObject(a1);
    context.Accounts.AddObject(a2);
    context.Accounts.AddObject(a3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var orders = from o in context.Orders
                 join a in context.Accounts on
                  new {Id = o.AccountId, City = o.ShipCity, State = o.ShipState }
                 equals
                  new {Id = a.AccountId, City = a.City, State = a.State }
                 select o;

    Console.WriteLine("Orders shipped to the account's city, state...");
    foreach (var order in orders)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    {
        Console.WriteLine("\tOrder {0} for {1}", order.AccountId.ToString(),
                          order.Amount.ToString("C"));
    }
}
```

The following is the output of the code in Listing 3-21:

```
Orders shipped to the account's city, state...

        Order 31 for $223.09

        Order 32 for $99.29
```

Licensed by Matt Allen 2175280

How It Works

To solve this problem, you could find all the accounts and then go through each Orders collection and find the orders that are in the same city and state as the account. For a small number of accounts, this may be a reasonable solution. But in general, it is best to push this sort of processing into the store layer where can be handled much more efficiently.

In the solution, we form the **join** by creating an anonymous type on each side of the **equals** clause. This is required when we join on more than one property. We need to make sure that both anonymous types are the same. They must have the same properties in the same order.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 4

Using Entity Framework in ASP.NET

In this chapter, we show you how to use the Entity Framework in your ASP.NET web pages. You could, of course, use many of the methods shown throughout this book in the code behind for your pages, but in this chapter we focus specifically on using the declarative approach provided by EntityDataSource and ObjectDataSource controls.

The EntityDataSource control together with the QueryExtender control provide a powerful, yet easy-to-understand way for you to build ASP.NET web pages that leverage much of the capabilities of the Entity Framework. The recipes in this chapter cover everything from simple searching to building a complete insert, update, delete, and search page. The last recipe in this chapter shows you how to use the ObjectDataSource control with Entity Framework.

Each of the recipes in this chapter starts with a new Empty ASP.NET Web Application. We've tried to keep things simple by not including all of extra code that comes with the default new ASP.NET Web Application template.

4-1. Building a Search Query

Problem

You want to declaratively build a search query in an ASP.NET page using EntityDataSource.

Solution

Let's say you have a model like the one in Figure 4-1.

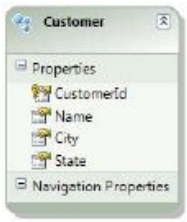


Figure 4-1. *A model with a Customer entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In Listing 4-1, we use three basic parts to build our search page: a table to structure the query parameters, a ListView to present the results, and an EntityDataSource to define the query. The code behind for page in Listing 4-2, simply populates the database with some test data in the **Page_Load()** event handler.

Listing 4-1. Using EntityDataSource to build a search query

```
<body>
 <form id="form1" runat="server">
   <div>
     <table>
       <tr>
         <td>Name</td>
         <td><asp:TextBox ID="Name" runat="server" /></td>
       </tr>
       <tr>
         <td>City</td>
         <td><asp:TextBox ID="City" runat="server" /></td>
       </tr>
       <tr>
         <td>State</td>
         <td><asp:TextBox ID="State" runat="server" /></td>
       </tr>
       <tr>
         <td colspan="2">
           <asp:Button ID="SearchCustomer" Text="Search" runat="server" />
         </td>
       </tr>
     </table>

     <asp:EntityDataSource ID="CustomerList" runat="server"
       ConnectionString="name=EFRecipesEntities"
       DefaultContainerName="EFRecipesEntities"
       Where="(@State is null || it.State = @State) &&
              (@City is null || it.City = @City) &&
              (@Name is null || it.Name LIKE '%' + @Name + '%')"
       EntitySetName="Customers">
       <WhereParameters>
         <asp:ControlParameter Name="Name" ControlID="Name" Type="String" />
         <asp:ControlParameter Name="City" ControlID="City" Type="String" />
         <asp:ControlParameter Name="State" ControlID="State" Type="String" />
       </WhereParameters>
     </asp:EntityDataSource>

     <asp:ListView ID="CustomerListView" runat="server"
       DataSourceID="CustomerList">
       <ItemTemplate>
         <tr>
           <td><%# Eval("Name") %></td>
           <td><%# Eval("City") %></td>
           <td><%# Eval("State") %></td>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        </tr>
      </ItemTemplate>
      <LayoutTemplate>
        <table>
          <tr>
            <th>Name</th>
            <th>City</th>
            <th>State</th>
          </tr>
          <tr id="ItemPlaceHolder" runat="server" />
        </table>
      </LayoutTemplate>
    </asp:ListView>
  </div>
 </form>
</body>
```

Listing 4-2. The code behind that builds the data to test our search page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        using (var context = new EFRecipesEntities())
        {
            // delete any previous data we might have
            context.ExecuteStoreCommand("delete from chapter4.customer");

            // insert some data
            context.Customers.AddObject(new Customer { Name = "Robin Rosen",
                                        City = "Olathe", State = "KS" });
            context.Customers.AddObject(new Customer { Name = "John Wise",
                                        City = "Springtown", State = "TX" });
            context.Customers.AddObject(new Customer { Name = "Karen Carter",
                                        City = "Raytown", State = "MO" });
            context.SaveChanges();
        }
    }
}
```

In the browser, the page looks something like the one in Figure 4-2.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

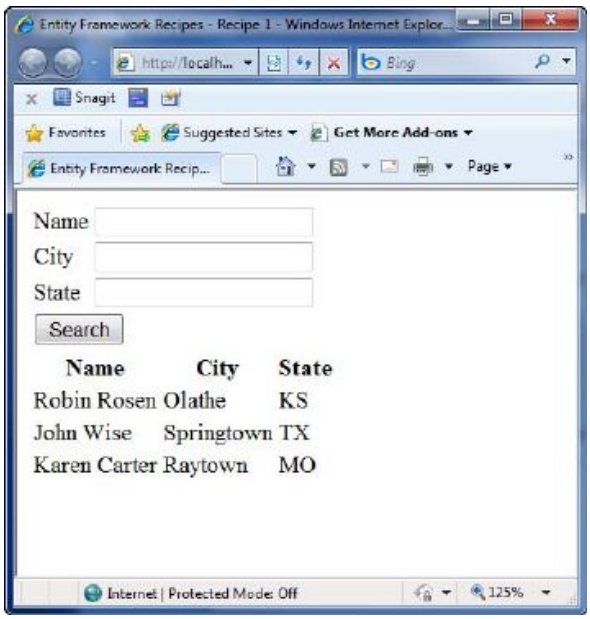


Figure 4-2. *The rendered page shown in a browser*

How It Works

In the first section of the page (refer to Listing 4-1), we format the query fields using a table. Nothing fancy here. The idea is to provide some structure to capture the three query fields: Name, City, and State. These values, or the lack of them, will be used in the EntityDataSource to form the filter for the query.

Next, we use an EntityDataSource to provide a data source for the results. In the EntityDataSource, we reference the connection string that was added to our web.config when the model was added to the project. In this recipe, the connection string is named EFRecipesEntities. In the EntityDataSource, we provide the entity state name for our query. In this case, the entity set name is Customers.

The Where attribute and the nested WhereParameters define the filter for our query. The Where attribute is set to a parameterized eSQL query. In the WhereParameters, we map the eSQL parameters to controls on the page. We map the @Name parameter to the TextBox with the Name ID, the @City parameter to the TextBox with the City ID, and the @State parameter to the TextBox with the State ID.

We use a ListView to display the results. The ListView is bound the EntityDataSource. In the ItemTemplate for the ListView, we format each Customer entity from the EntityDataSource as a row in a table. In the LayoutTemplate, we provide the placement of the items inside the well-formed table along with the header row.

The code behind, shown in Listing 4-2, is used just to populate the database with some usable test data. Here we do all the work in the `Page_Load()` event. We start off by deleting any rows that might be in the database. Next, we populate the database with a few Customers.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4.2. Building CRUD Operations in an ASP.NET Web Page

Problem

You want to build an ASP.NET page that allows inserting, updating, deleting, and reading from your model.

Solution

Let's say you have an application that manages the membership in a local club. You have a model like the one in Figure 4-3.

Figure 4-3. *A model with a Member entity that contains a member's name and email address*

The model contains a Member entity. You want to create a simple ASP.NET page shows all the club members and allows the user to create a new member, update an existing member, and delete a member. There are lots of ways to do this, but you want to use an EntityDataSource control to do as much of the work declaratively as possible.

To create this page, we'll need three basic parts. First, we'll need a way to show all the club members. We'll use a ListView to do this. Next, we'll need a way to populate the ListView will the club member. We'll use an EntityDataSource for this. Finally, our club may grow large enough that we can't reasonably display all the members on a single web page. We will need a way to break up the display into multiple pages. We'll use a DataPager control to do this.

The code for the ASP.NET page is shown in Listing 4-3. We'll also need some code behind to handle a few events. The code behind is shown in Listing 4-4.

Listing 4-3. The ASP.NET page with the three core parts: ListView, EntityDataSource, and DataPager

```
<body>
  <form id="form1" runat="server">
    <div style="font-size:larger; margin: 10px;">Manage Club Members</div>
    <div>
      <asp:Button ID="Insert" Text="Insert New Member" runat="server"
                  OnClick="InsertMember" />
      <asp:ListView ID="membersList" runat="server"
               DataSourceID="membersDataSource"
               DataKeyNames="MemberId" OnItemInserted="membersList_ItemInserted">
        <LayoutTemplate>
          <table>
            <tr>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        <th colspan="2"> </th>
        <th>Name</th>
        <th>Email</th>
      </tr>
      <tr id="itemPlaceholder" runat="server" />
    </table>
  </LayoutTemplate>
  <ItemTemplate>
    <tr>
      <td><asp:LinkButton Text="Delete" CommandName="Delete"
              runat="server" /></td>
      <td><asp:LinkButton Text="Edit" CommandName="Edit"
              runat="server" /></td>
      <td><%# Eval("Name") %></td>
      <td><%# Eval("Email") %></td>
    </tr>
  </ItemTemplate>
  <InsertItemTemplate>
    <tr>
      <td colspan="4">
        <table>
          <tr>
            <td>Name:</td>
            <td><asp:TextBox ID="Name" runat="server"
                      Text='<%# Bind("Name") %>' /></td>
          </tr>
          <tr>
            <td>Email:</td>
            <td><asp:TextBox ID="Email" runat="server"
                      Text='<%# Bind("Email") %>' /></td>
          </tr>
          <tr>
            <td colspan="2">
              <asp:Button Text="Insert" CommandName="Insert"
                    runat="server" /> 
              <asp:Button Text="Cancel" CommandName="Cancel"
                    OnClick="CancelClick" runat="server" />
            </td>
          </tr>
        </table>
      </td>
    </tr>
  </InsertItemTemplate>

  <EditItemTemplate>
    <tr>
      <td colspan="4">
        <table>
          <tr>
            <td>Name:</td>
            <td><asp:TextBox ID="Name" runat="server"
                      Text='<%# Bind("Name") %>' /></td>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                </tr>
                <tr>
                  <td>Email:</td>
                  <td><asp:TextBox ID="Email" runat="server"
                           Text='<%# Bind("Email") %>' /></td>
                </tr>
                <tr>
                  <td colspan="2">
                    <asp:Button Text="Update" CommandName="Update"
                           runat="server" />  
                    <asp:Button Text="Cancel" CommandName="Cancel"
                           runat="server" />
                  </td>
                </tr>
              </table>
            </td>
          </tr>
        </EditItemTemplate>
      </asp:ListView>

      <asp:EntityDataSource ID="membersDataSource" runat="server"
            ConnectionString="name=EFRecipesEntities"
            DefaultContainerName="EFRecipesEntities"
            EnableInsert="true" EnableUpdate="true" EnableDelete="true"
            EntitySetName="Members" />

      <asp:DataPager ID="Pager" runat="server" PagedControlID="membersList"
            PageSize="2">
        <Fields>
          <asp:NumericPagerField ButtonCount="10" />
        </Fields>
      </asp:DataPager>
    </div>
  </form>
</body>
```

Listing 4-4. The code behind that handles the events for our ASP.NET page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!Page.IsPostBack)
        {
            using (var context = new EFRecipesEntities())
            {
                context.ExecuteStoreCommand("delete from chapter4.Member");
                context.Members.AddObject(new Member { Name = "Robert Dewey",
                                                Email = "RobertD@gmail.com" });
                context.Members.AddObject(new Member { Name = "Nancy Steward",
                                                Email = "NSteward@AOL.com" });
                context.Members.AddObject(new Member { Name = "Robin Rosen",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                                Email = "RRosen@Regenix.com" });
                context.SaveChanges();
            }
        }
    }

    protected void membersList_ItemInserted(object sender,
                                            ListViewInsertedEventArgs e)
    {
        if (e.Exception == null)
        {
            membersList.InsertItemPosition = InsertItemPosition.None;
        }
    }

    protected void CancelClick(object sender, EventArgs e)
    {
        membersList.InsertItemPosition = InsertItemPosition.None;
    }

    protected void InsertMember(object sender, EventArgs e)
    {
        membersList.InsertItemPosition = InsertItemPosition.FirstItem;
    }
}
```

The page in Listing 4-3 and the code in Listing 4-4 displays a page that lists the club members, along with buttons for inserting new members as well as editing and deleting current members. The listing page is shown in Figure 4-4. The insert page is shown in Figure 4-5. The edit page is shown in Figure 4-6.

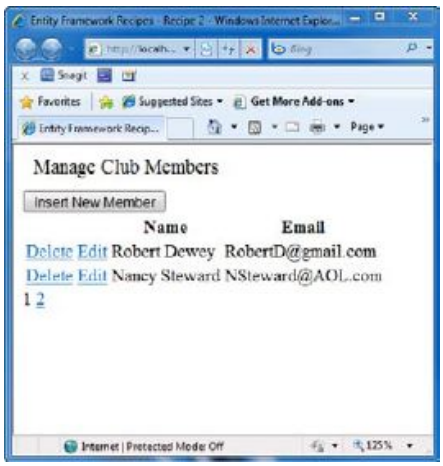


Figure 4-4. *The listing of the club members. The data pager at the bottom allows the user to move forward and backward through the pages containing members.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

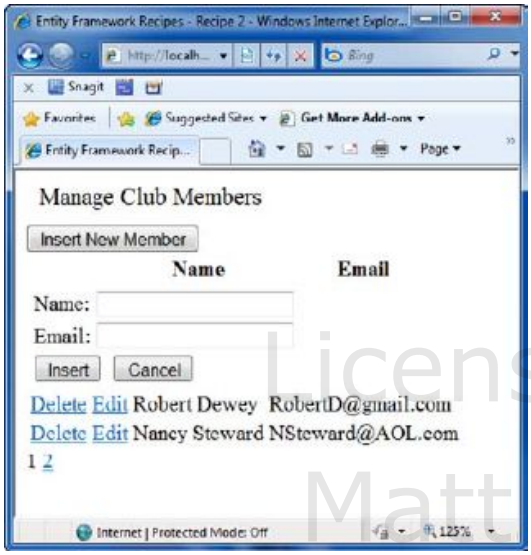


Figure 4-5. *Inserting a new member. The TextBoxes allow the user to enter the member information. Clicking the Insert button causes the new record to be added to the database.*

Licensed by Matt Allen 2175280

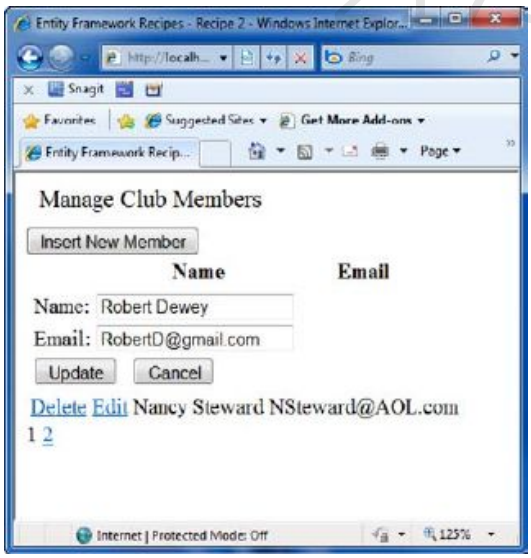


Figure 4-6. *Clicking the Edit button on a member shows this view allowing editing of the member.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

The code in Listing 4-3 can be broken up into three parts: the ListView that handles much of the user interface, the EntityDataSource that handles connection to the model, and the DataPager that provides paging for the members.

The ListView is the largest part of the code and is the heart of the user interface. In the ListView, we use the DataSourceID attribute to bind to the data from our model via the EntityDataSource. We use the DataKeyNames attribute to indicate the MemberId property that contains the key for our Members. The various templates allow us to provide the layout for the listing, editing, and inserting layouts.

The ItemTemplate provides our read-only, listing view of the members. Notice that we introduce two buttons on each row. One button triggers editing of the member, and the other button triggers deleting the member. The CommandName attributes for the buttons trigger the actions. If the user clicks the Edit link button, the view is switched to the EditItemTemplate.

In the EditItemTemplate, we show a TextBox to edit the Name property and a TextBox to edit the Email property. We use **Bind()** to retrieve the current value from the selected entity and push the changed value to the property.

Just like with the EditItemTemplate, in the InsertItemTemplate we show a TextBox for the Name property value and a TextBox for the Email property value. We also use **Bind()** to push the values to the new Member entity.

The EntityDataSource control connects the model to the ListView control. In the EntityDataSource control, we enabled inserting, deleting, and updating. The ConnectionString and EntitySetName attributes bind this EntityDataSource control to our model and Members entity set within the model.

Finally, the DataPager control allows us to show a set number of members per page. This makes the user interface much more manageable as the number of club members grows. Here we've limited a page to just two members to keep things short.

The ListView control is tied to the EntityDataSource control through IDs. The ListView control's DataSourceID is the EntityDataSource's ID. The DataPager's PagedControlID is the ID of the ListView.

There are a handful of events that we hand in the code behind in Listing 4-4. In the **Page_Load()** event, we delete any previous data from the database and populate it with our initial test data. Of course, you wouldn't normally do this, but it makes it easy to demonstrate what's going on. For the other events, we revert back to the listing view when an insert is successful or when the user clicks Cancel. When the Insert New Member button is clicked, we handle the event by showing in the InsertItemTemplate.

4-3. Executing Business Logic When Changes Are Saved

Problem

You are using an EntityDataSource control and you want to make sure that your business logic is executed inside the **SavingChanges** event.

Solution

Let's say our model looks something like the one in Figure 4-7.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 4-7. *A model with a single PurchaseOrder entity*

We have a purchase order with a company name and an amount. We want the CreateDate property to be set automatically. To do this, we need to intercept the **SavingChanges** event and set the CreateDate property to the current date and time if the entity is newly added. We want to do this while using an EntityDataSource control on our page.

To do this, follow the pattern in Listings 4-5 and 4-6.

Listing 4-5. The code for the ASP.NET page that captures the Company name and Amount

```
<body>
    <form id="form1" runat="server">
    <div>
      <asp:DetailsView ID="detailsView" runat="server"
         AutoGenerateRows="false" DataSourceID="orderSource"
         DefaultMode="Insert">
        <Fields>
          <asp:BoundField DataField="Company" HeaderText="Company" />
          <asp:BoundField DataField="Amount" HeaderText="Amount" />
          <asp:CommandField ShowInsertButton="true" />
        </Fields>
      </asp:DetailsView>

      <asp:EntityDataSource ID="orderSource" runat="server"
          ConnectionString="name=EFRecipesEntities"
          ContextTypeName="Recipe3.EFRecipesEntities"
          DefaultContainerName="EFRecipesEntities"
          EnableInsert="true" EntitySetName="PurchaseOrders" />
    </div>
    </form>
</body>
```

Listing 4-6. The code behind for the page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges += (o, e) =>
            {
                var orders = this.ObjectStateManager
                    .GetObjectStateEntries(System.Data.EntityState.Added)
                    .Select(en => en.Entity as PurchaseOrder);
                foreach (var order in orders)
                {
                    order.CreateDate = DateTime.Now;
                }
            };
    }
}
```

The resulting page is shown in a browser in Figure 4-8.

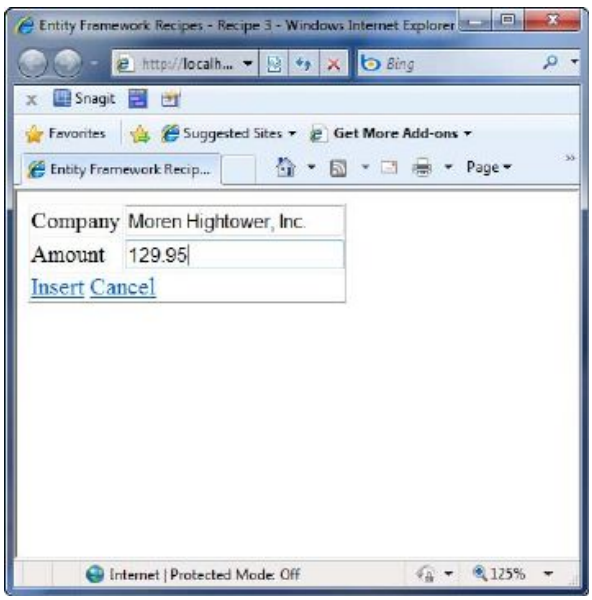


Figure 4-8. *The simple input form used to capture the Company name and Amount*

How It Works

In the code behind in Listing 4-6, we use the partial method **OnContextCreated()** to wire in our event handler for the **SavingChanges** event. In our handler, we gather up all the PurchaseOrder entities that are in the added state and assign the current date and time to the CreateDate property.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4-4. Loading Related Entities

Problem

You are using EntityDataSource in your ASP.NET page and you want to load related entities.

Solution

Suppose you have a model like the one in Figure 4-9.

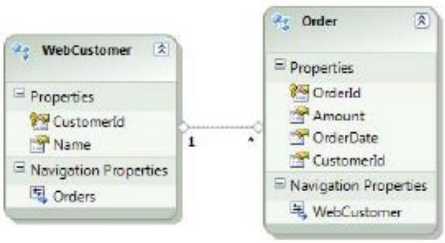


Figure 4-9. *A model for a web customer and her orders*

In our model, each web customer can have many orders. We want to use an EntityDataSource control to load the orders and include the customer associated with each of the orders.

To eagerly load the customer associated with each order, use the Include attribute on the EntityDataSource control, as illustrated in Listing 4-7.

Listing 4-7. The ASP.NET page to display our customer's orders

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ListView ID="orderslist" runat="server" DataSourceId="orders">
        <LayoutTemplate>
          <table>
            <tr>
              <th>Name</th>
              <th>Amount</th>
              <th>OrderDate</th>
            </tr>
            <tr id="itemPlaceHolder" runat="server" />
          </table>
        </LayoutTemplate>
        <ItemTemplate>
          <tr>
            <td><%# Eval("WebCustomer.Name") %></td>
            <td><%# Eval("Amount") %></td>
            <td><%# Eval("OrderDate") %></td>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            </tr>
          </ItemTemplate>
        </asp:ListView>
        <asp:EntityDataSource ID="orders" runat="server"
             DefaultContainerName="EFRecipesEntities" Include="WebCustomer"
             ConnectionString="name=EFRecipesEntities" EntitySetName="Orders" />
    </div>
  </form>
</body>
```

In the code behind in Listing 4-8, we handle the **Page_Load** event by deleting any previous test data and populating the WebCustomers and Orders with fresh test data.

Listing 4-8. The code behind for our ASP.NET page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        using (var context = new EFRecipesEntities())
        {
            context.ExecuteStoreCommand("delete from chapter4.[order]");
            context.ExecuteStoreCommand("delete from chapter4.webcustomer");

            var cust1 = new WebCustomer { Name = "Joan Steward" };
            var cust2 = new WebCustomer { Name = "Allen Colbert" };
            var cust3 = new WebCustomer { Name = "Phil Marlowe" };
            var order1 = new Order { Amount = 29.95M,
                 OrderDate = DateTime.Parse("3/18/2010") };
            var order2 = new Order { Amount = 84.99M,
                 OrderDate = DateTime.Parse("3/20/2010") };
            var order3 = new Order { Amount = 99.95M,
                 OrderDate = DateTime.Parse("4/10/2010") };
            order1.WebCustomer = cust1;
            order2.WebCustomer = cust2;
            order3.WebCustomer = cust3;
            context.Orders.AddObject(order1);
            context.Orders.AddObject(order2);
            context.Orders.AddObject(order3);
            context.SaveChanges();
        }
    }
}
```

The resulting page is shown in a browser in Figure 4-10.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

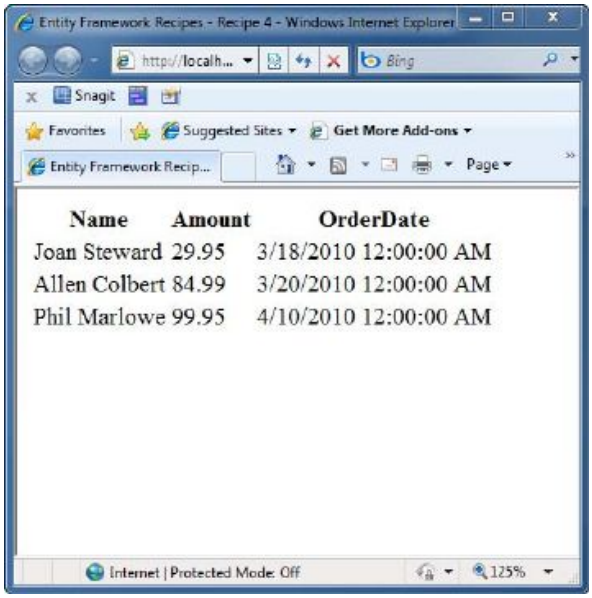


Figure 4-10. *Web customers with their orders*

How It Works

By default, Entity Framework does not load the related entities like our WebCustomer. To eagerly load them when using an EntityDataSource control, use the Include attribute and provide the path through the navigation properties of all the related entities you want loaded.

4-5. Searching with QueryExtender

Problem

You want to use a QueryExtender control with an EntityDataSource control to implement searching in your ASP.NET page.

Solution

Suppose you have a model like the one in Figure 4-11.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

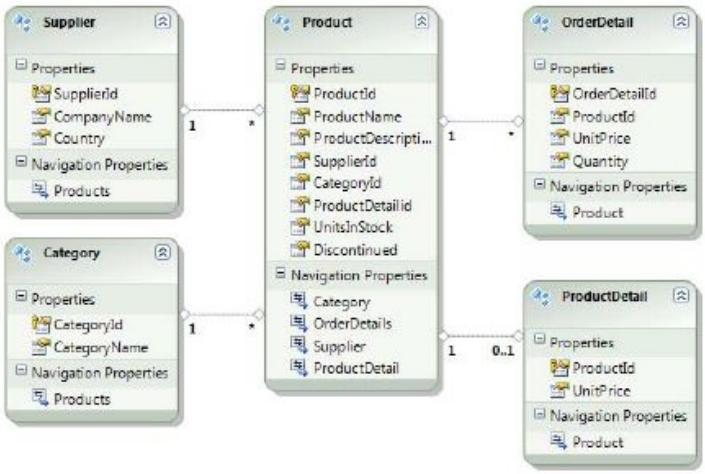


Figure 4-11. *A model for products, suppliers, and orders*

In our model, a product has a supplier, is in a category, and may have some orders. We want to build an ASP.NET page using an EntityDataSource control and a QueryExtender control to search our model across these related entities.

The QueryExtender control provides a good deal of flexibility in formulating a search query. We want to map TextBoxes for a number of properties to the QueryExtender control to build a query. We want to display the results of the query in a grid. To do this, follow the pattern in Listing 4-9.

Listing 4-9. The ASP.NET search page

```
<body>
  <form id="form1" runat="server">
    <div>
      <table>
        <tr>
          <td>Name or Description</td>
          <td><asp:TextBox ID="ProductName" runat="server" /></td>
        </tr>
        <tr>
          <td>Discontinued</td>
          <td>
            <asp:DropDownList ID="Discontinued" runat="server">
              <asp:ListItem Text="All" Value="" />
              <asp:ListItem Text="Yes" Value="true" />
              <asp:ListItem Text="No" Value="false" />
            </asp:DropDownList>
          </td>
        </tr>
        <tr>
          <td>Category</td>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    <td><asp:TextBox ID="CategoryName" runat="server" /></td>
  </tr>
  <tr>
    <td>Units In Stock</td>
    <td><asp:TextBox ID="UnitsInStock" runat="server" /></td>
  </tr>
  <tr>
    <td>Price From</td>
    <td><asp:TextBox ID="FromPrice" runat="server" />   
        Price To <asp:TextBox ID="ToPrice" runat="server" />
    </td>
  </tr>
  <tr>
    <td>Supplier Country</td>
    <td><asp:TextBox ID="SupplierCountry" runat="server" /></td>
  </tr>
  <tr>
    <td>Total Sales</td>
    <td><asp:TextBox ID="TotalSales" runat="server" /></td>
  </tr>
  <tr>
    <td align="left" colspan="2" > <asp:Button ID="SearchButton"
        Text="Search" runat="server" /></td>
  </tr>
</table>

<asp:GridView ID="GridView1" runat="server" AllowPaging="true"
     PageSize="50" AutoGenerateColumns="false" DataSourceID="DataSource">
  <Columns>
    <asp:BoundField DataField="ProductName" HeaderText="Product Name" />
    <asp:BoundField DataField="ProductDescription"
          HeaderText="Product Description" />
    <asp:CheckBoxField DataField="Discontinued" HeaderText="Discontinued" />
    <asp:TemplateField HeaderText="UnitPrice">
      <ItemTemplate><%# Eval("ProductDetail.UnitPrice","{0:C}") %>
      </ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="CategoryName">
      <ItemTemplate><%# Eval("Category.CategoryName") %></ItemTemplate>
    </asp:TemplateField>
    <asp:BoundField DataField="UnitsInStock" HeaderText="Units In Stock" />
    <asp:TemplateField HeaderText="Supplier Country">
      <ItemTemplate><%# Eval("Supplier.Country") %></ItemTemplate>
    </asp:TemplateField>
    <asp:TemplateField HeaderText="Total Sales">
      <ItemTemplate><%# Eval("TotalSales","{0:C}") %></ItemTemplate>
    </asp:TemplateField>
  </Columns>
</asp:GridView>

<asp:EntityDataSource ID="DataSource" runat="server"
    ConnectionString="name=EFRecipesEntities"
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            Include="Category,ProductDetail,Supplier,OrderDetails"
            DefaultContainerName="EFRecipesEntities"
           EnableFlattening="false" EntitySetName="Products" />

        <asp:QueryExtender ID="QueryExtender1" runat="server"
            TargetControlID="DataSource">
          <asp:SearchExpression SearchType="Contains"
               DataFields="ProductName,ProductDescription">
            <asp:ControlParameter ControlID="ProductName" />
          </asp:SearchExpression>
          <asp:OrderByExpression DataField="UnitsInStock" Direction="Descending">
            <asp:ThenBy DataField="ProductDetail.UnitPrice"
                 Direction="Ascending" />
          </asp:OrderByExpression>
          <asp:PropertyExpression>
            <asp:ControlParameter Name="Discontinued" ControlID="Discontinued" />
            <asp:ControlParameter Name="UnitsInStock" ControlID="UnitsInStock" />
            <asp:ControlParameter Name="Supplier.Country"
                ControlID="SupplierCountry" />
          </asp:PropertyExpression>
          <asp:RangeExpression DataField="ProductDetail.UnitPrice"
                MinType="Inclusive" MaxType="Exclusive">
            <asp:ControlParameter ControlID="FromPrice" />
            <asp:ControlParameter ControlID="ToPrice" />
          </asp:RangeExpression>
          <asp:CustomExpression OnQuerying="ProductsWithCategory">
            <asp:ControlParameter Name="CategoryName" ControlID="CategoryName" />
          </asp:CustomExpression>
          <asp:MethodExpression MethodName="ProductWithSalesGreaterThan">
            <asp:ControlParameter Name="TotalSales" Type="Decimal"
                 ControlID="TotalSales" />
          </asp:MethodExpression>
        </asp:QueryExtender>
      </div>
    </form>
</body>
```

Listing 4-10. The code behind for our search page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        using (var context = new EFRecipesEntities())
        {
            // cleanup from previous tests
            context.ExecuteStoreCommand("delete from chapter4.productdetail");
            context.ExecuteStoreCommand("delete from chapter4.orderdetail");
            context.ExecuteStoreCommand("delete from chapter4.product");
            context.ExecuteStoreCommand("delete from chapter4.category");
            context.ExecuteStoreCommand("delete from chapter4.supplier");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        // add in our test data
        var s1 = new Supplier { CompanyName = "Backcountry Supply",
                                Country = "USA" };
        var s2 = new Supplier { CompanyName = "Alpine Tent",
                                Country = "Italy" };
        var s3 = new Supplier { CompanyName = "Ace Footware",
                                Country = "USA" };
        var c1 = new Category { CategoryName = "Tents" };
        var c2 = new Category { CategoryName = "Shoes/Boots" };
        var pd1 = new ProductDetail { UnitPrice = 99.95M };
        var pd2 = new ProductDetail { UnitPrice = 129.95M };
        var pd3 = new ProductDetail { UnitPrice = 39.95M };
        var p1 = new Product { ProductName = "Pup Tent",
                        ProductDescription = "Small and packable tent",
                        Discontinued = true, UnitsInStock = 4 };
        var p2 = new Product { ProductName = "Trail Boot",
                        ProductDescription = "Perfect boot for hiking",
                        Discontinued = false, UnitsInStock = 19 };
        var p3 = new Product { ProductName = "Family Tent",
                        ProductDescription = "Sleeps 2 adults + 2 children",
                        Discontinued = false, UnitsInStock = 10 };
        var od1 = new OrderDetail { UnitPrice = 39.95M, Quantity = 1};
        var od2 = new OrderDetail { UnitPrice = 129.95M, Quantity = 2 };
        var od3 = new OrderDetail { UnitPrice = 99.95M, Quantity = 1 };
        p1.Supplier = s2;
        p1.Category = c1;
        p1.ProductDetail = pd3;
        p1.OrderDetails.Add(od1);
        p2.Supplier = s3;
        p2.Category = c2;
        p2.OrderDetails.Add(od2);
        p2.ProductDetail = pd2;
        p3.Supplier = s1;
        p3.Category = c1;
        p3.ProductDetail = pd1;
        p3.OrderDetails.Add(od3);
        context.Products.AddObject(p1);
        context.Products.AddObject(p2);
        context.Products.AddObject(p3);
        context.SaveChanges();
    }
}

protected void ProductsWithCategory(object sender,
                                    CustomExpressionEventArgs e)
{
    if (e.Values["CategoryName"] != null)
    {
        var catnames = e.Values["CategoryName"].ToString().Split(',');
        e.Query = from p in e.Query.Cast<Product>()
                  where catnames.Contains(p.Category.CategoryName)
                  select p;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }
    }

    static public IQueryable<Product> ProductWithSalesGreaterThan(
                                IQueryable<Product> query,
                                decimal totalSales)
    {
        return from p in query
               where p.OrderDetails
                      .Sum(od => od.UnitPrice * od.Quantity) > totalSales
               select p;
    }
}

public partial class Product
{
    public decimal TotalSales
    {
        get
        {
            return this.OrderDetails.Sum(od => od.UnitPrice * od.Quantity);
        }
    }
}
```

The resulting search page is shown in Figure 4-12. The user can enter the search parameters in any or all of the TextBoxes to filter the results displayed in the grid.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

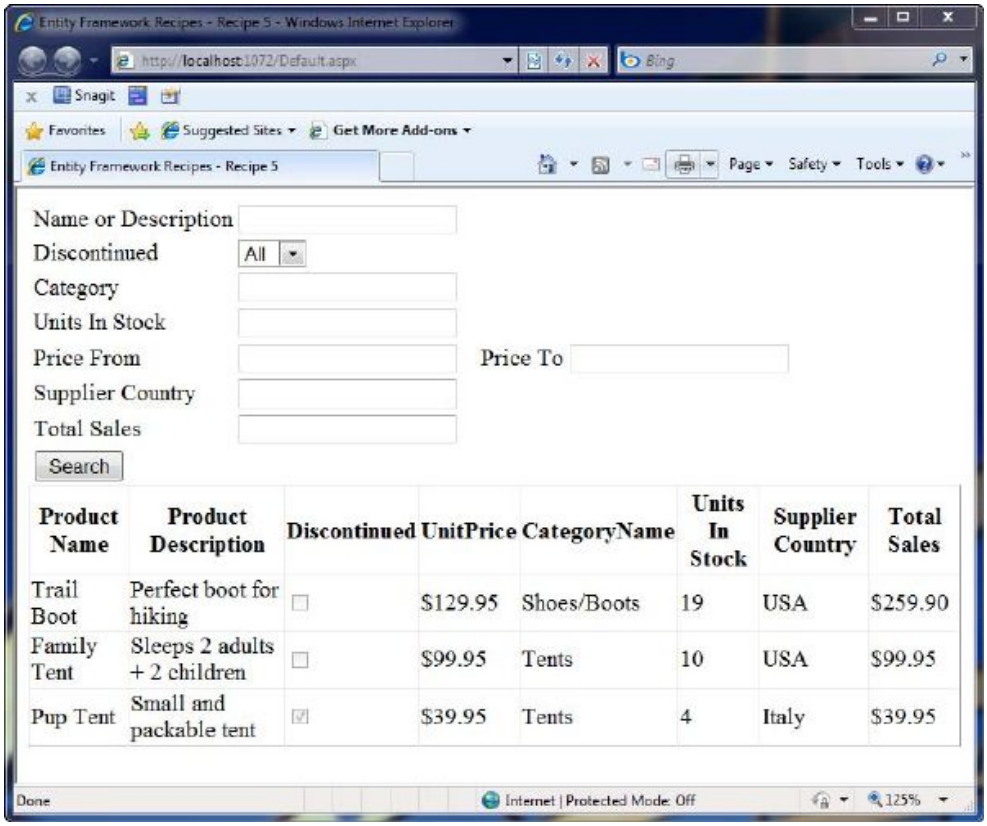


Figure 4-12. *The completed search page implemented with an EntityDataSource control and a QueryExtender control*

How It Works

The model is a little involved, and the code in Listings 4-9 and 4-10 seem to reflect this. Let's break down the ASP.NET page part by part.

The first part of Listing 4-9 is a table providing structure for the properties we want to use for searching. There's nothing too complex here, just a bunch of TextBoxes and a DropDownList formatted in a table.

The next part is a GridView control. This provides the structure for the results. We bind the control to the EntityDataSource control through the DataSourceID attribute. The columns are mapped to properties of the objects returned by the EntityDataSource control. This is not much different from the binding we've seen in the previous recipes in this chapter. There is one difference to note here: we're binding to a TotalSales property that is not present in our model. This property represents the total sales

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

for the product. To get this value, we added a TotalSales property to the Product entity in the code behind shown in Listing 4-10.

As with the previous recipes, we use the EntityDataSource control to load the data. We've use the Include attribute to eagerly load the Category, ProductDetail, Supplier, and OrderDetails navigation properties. This causes the related entities to be loaded along with each Product entity.

And finally, we have the QueryExtender control. This control provides the filtering that we need to implement searching. We set the TargetControlID to the ID of the EntityDataSource control. This ties the QueryExtender control to our EntityDataSource control. We are exercising quite a few expression types in our QueryExtender control. Each of these is used to extend the query.

In the SearchExpression, we set the SearchType to Contains and map the DataFields to the ProductName and ProductDescription properties. The expression will get its filter value from the ProductName TextBox. When not empty, this expression will filter the result set to product's that contain the string in the ProductName TextBox in either the ProductName or ProductDescription properties.

The OrderByExpression orders the result set first by UnitsInStock and then by UnitPrice.

The PropertyExpression filters the result set by the Discontinued, UnitsInStock, and the Supplier's country. The corresponding TextBoxes are mapped through their IDs.

The RangeExpression is used to filter the result set by a range of values for the UnitPrice property. We denote this in our search TextBoxes with the Price From and Price To fields.

We use the CustomExpression to add on our own arbitrary query to the one built by the QueryExtender control. We've implemented this in the `ProductsWithCategory()` method. Here we additionally filter by products that are in the given category.

With the MethodExpression we use a method outside of our class to perform additional filtering. The method is passed an `IQueryable<Product>` and a total sales threshold and returns an `IQueryable<Product>` that filters by the threshold. This is implemented in Listing 4-10 in the `ProductWithSalesGreaterThan()` method.

The result of all of this is the search page shown in Figure 4-12. Nearly all the logic is implemented declaratively in the ASP.NET page.

4-6. Retrieving a Derived Type Using an EntityDataSource Control

Problem

You want to load a derived type from your Table per Hierarchy inheritance model using an EntityDataSource control.

Solution

Suppose you have a model like the one in Figure 4-13.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

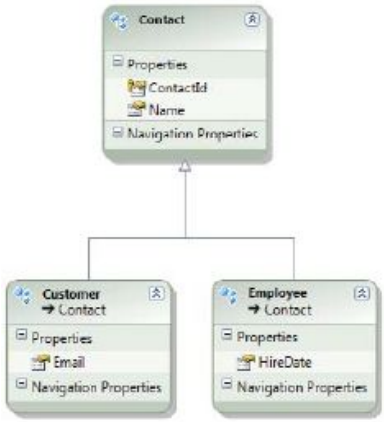


Figure 4-13. *A model using Table per Hierarchy inheritance with derived types Customer and Employee*

The model in Figure 4-13 uses Table per Hierarchy inheritance with Customer and Employee as derived types. The discriminator values for the derived types are the strings "Customer" and "Employee."

To filter the result set to a specific derived type using an EntityDataSource control, name the type using the EntityTypeFilter attribute. The code in Listings 4-11 and 4-12 illustrate retrieving both the Employees and Customers using an EntityDataSource control. The resulting page is shown in Figure 4-14.

Listing 4-11. The ASP.NET page using an EntityDataSource to retrieve derived types

```
<body>
    <form id="form1" runat="server">
    <div>
      <h2>Employees</h2>
      <asp:GridView ID="GridView1" runat="server" DataSourceID="EmployeesSource"
              AutoGenerateColumns="true" />
      <asp:EntityDataSource ID="EmployeesSource" runat="server"
              ConnectionString="name=EFRecipesEntities"
              DefaultContainerName="EFRecipesEntities" EnableFlattening="false"
              EntitySetName="Contacts" EntityTypeFilter="Employee" />

      <h2>Customers</h2>
      <asp:GridView ID="GridView2" runat="server" DataSourceID="CustomersSource"
              AutoGenerateColumns="true" />
      <asp:EntityDataSource ID="CustomersSource" runat="server"
              ConnectionString="name=EFRecipesEntities"
              DefaultContainerName="EFRecipesEntities" EnableFlattening="false"
              EntitySetName="Contacts" EntityTypeFilter="Customer" />
    </div>
    </form>
</body>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 4-12. The code behind for the page

```
protected void Page_Load(object sender, EventArgs e)
{
    using (var context = new EFRecipesEntities())
    {
        // delete the previous test data
        context.ExecuteStoreCommand("delete from chapter4.contact");

        // insert some new test data
        context.Contacts.AddObject(new Customer { Name = "Joan Ryan",
                                                  Email = "joanr@gmail.com" });
        context.Contacts.AddObject(new Customer { Name = "Robert Kelly",
                                                  Email = "rkelly@gmail.com" });
        context.Contacts.AddObject(new Employee { Name = "Karen Stanford",
                                                  HireDate = DateTime.Parse("1/21/2010")});
        context.Contacts.AddObject(new Employee { Name = "Phil Marlowe",
                                                  HireDate = DateTime.Parse("2/12/2009") });
        context.SaveChanges();
    }
}
```

The resulting web page as rendered in a browser is shown in Figure 4-14.

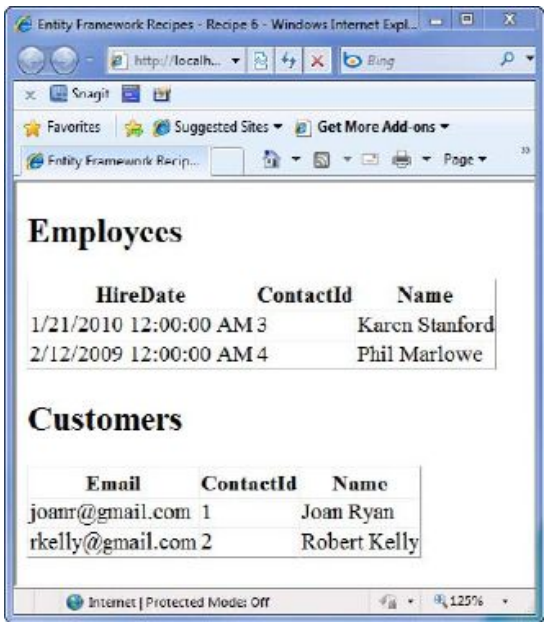


Figure 4-14. *The web page showing the properties of the derived entities Employee and Customer*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

The code in Listing 4-11 for the ASP.NET page uses an EntityDataSource control to load instances of a specific derived type and a GridView control to display the result set. We do this for the Employee derived type and the Customer derived type.

The code behind in Listing 4-12 deletes the previous test data and populates the model with the new test data. This is done in the **Page_Load()** event handler.

4-7. Filtering with ASP.NET's URL Routing

Problem

You want to simplify the URLs on your site using a RouteTable and want to leverage these routes to filter the result sets from an EntityDataSource control.

Solution

Suppose your model looks like the one in Figure 4-15.

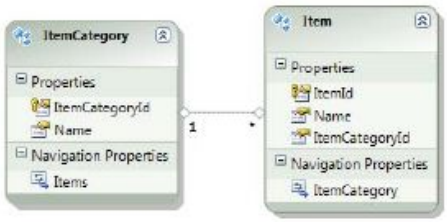


Figure 4-15. *A model for items and their categories*

In Figure 4-15, we've modeled our products, represented here by the Item entity, together with their categories. On a typical eCommerce website, we would show products by category. We want to avoid exposing query strings like "/Product.aspx?Category=Tents" in our URLs. These cryptic URLs simplify programming a little, but don't help us much when it comes to search engine optimization. We would rather have URLs that look more like "/Products/Tents". We can get this more SEO-friendly URL structure by using routing.

Routes are typically created in the **Application_Start()** event handler in Global.asax. The code in Listing 4-13 illustrates adding a route for our Products.aspx page.

Listing 4-13. Adding the Route in Global.asax

```
protected void Application_Start(object sender, EventArgs e)
{
    RouteTable.Routes.MapPageRoute("Products", "Products/{category}",
                                   "~/Products.aspx");
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In our Products.aspx, we use the category name bound to the "category" parameter in a QueryExtender control as illustrated in Listing 4-14. We use the code behind in Listing 4-15 to clear out any previous test data and populate our model with fresh test data. Figures 4-16 and 4-17 show the rendered pages for categories Tents and Cooking Equipment.

Listing 4-14. The Products.aspx page that displays the products filtered by category

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="false"
             DataSourceID="itemSource">
        <Columns>
          <asp:BoundField DataField="Name" HeaderText="Product" />
          <asp:TemplateField HeaderText="Category">
            <ItemTemplate><%# Eval("ItemCategory.Name") %></ItemTemplate>
          </asp:TemplateField>
        </Columns>
      </asp:GridView>

      <asp:EntityDataSource ID="itemSource" runat="server"
            EntitySetName="Items" Include="ItemCategory"
            ConnectionString="name=EFRecipesEntities"
            DefaultContainerName="EFRecipesEntities" />
      <asp:QueryExtender ID="search" TargetControlID="itemSource" runat="server">
        <asp:PropertyExpression>
          <asp:RouteParameter Name="ItemCategory.Name" RouteKey="category" />
        </asp:PropertyExpression>
      </asp:QueryExtender>
    </div>
  </form>
</body>
```

Listing 4-15. The code behind that populates the model with the test data

```
public partial class Products : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        using (var context = new EFRecipesEntities())
        {
            // delete any previous test data
            context.ExecuteStoreCommand("delete from chapter4.item");
            context.ExecuteStoreCommand("delete from chapter4.itemcategory");

            // populate with some test data
            var cat1 = new ItemCategory { Name = "Tents" };
            var cat2 = new ItemCategory { Name = "Cooking Equipment" };
            context.Items.AddObject(new Item { Name = "Backpacking Tent",
                                               ItemCategory = cat1 });
            context.Items.AddObject(new Item { Name = "Camp Stove",
                                               ItemCategory = cat2 });
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            context.Items.AddObject(new Item { Name = "Dutch Oven",
                                               ItemCategory = cat2 });
            context.Items.AddObject(new Item { Name = "Alpine Tent",
                                               ItemCategory = cat1 });
            context.Items.AddObject(new Item { Name = "Fire Starter",
                                               ItemCategory = cat2 });
            context.SaveChanges();
        }
    }
}
```

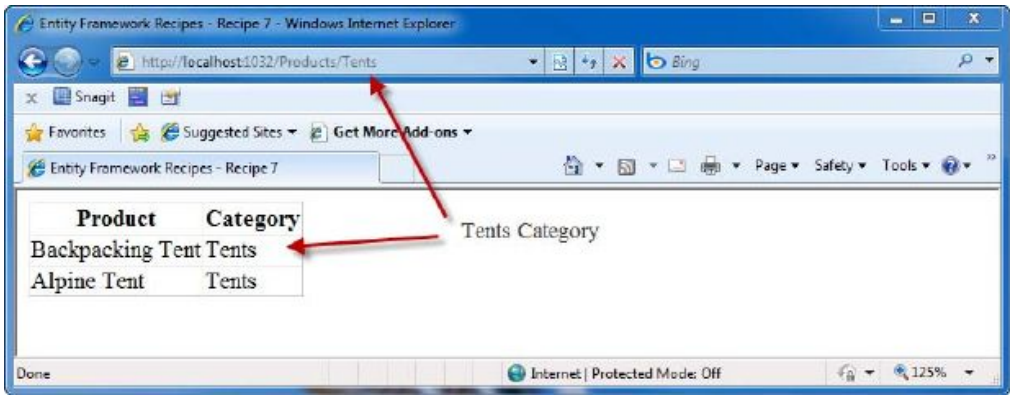


Figure 4-16. *Using the route /Products/Tents, the result set is filtered to the "Tents" category.*

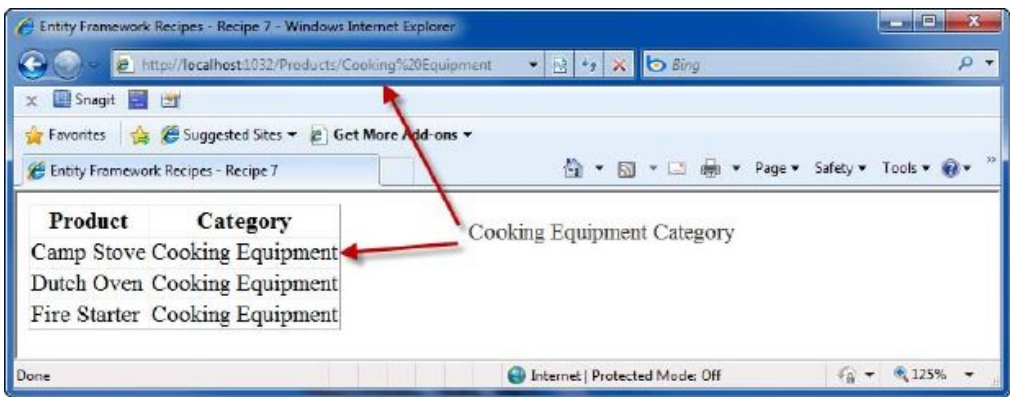


Figure 4-17. *Using the route /Products/Cooking Equipment, the result set is filtered to the "Cooking Equipment" category.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

In the **Application_Start()** event handler in Global.asax, we mapped the route /Products/{category} to the ~/Products.aspx page. The route key, category, is bound to the actual category string in the URL. In the QueryExtender control in Products.aspx, we used the category route key in a RouteParameter to filter the result set to just those products in the given category.

If you need more control over the filtering or don't want to use a QueryExtender control, you can use the OnQueryCreated attribute on the EntityDataSource control to inject your own filter on the result set.

In Listings 4-16 and 4-17, we have the same GridView as in Listing 4-14, but we have replaced the QueryExtender control with our own OnQueryCreated handler.

Listing 4-16. The same products page but without the QueryExtender control

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:GridView ID="GridView1" runat="server" AutoGenerateColumns="false"
            DataSourceID="itemSource">
        <Columns>
          <asp:BoundField DataField="Name" HeaderText="Product" />
          <asp:TemplateField HeaderText="Category">
            <ItemTemplate><%# Eval("ItemCategory.Name") %></ItemTemplate>
          </asp:TemplateField>
        </Columns>
      </asp:GridView>

      <asp:EntityDataSource ID="itemSource" runat="server" EntitySetName="Items"
            Include="ItemCategory" ConnectionString="name=EFRecipesEntities"
            DefaultContainerName="EFRecipesEntities"
            OnQueryCreated="ProdFilter" />
    </div>
  </form>
</body>
```

Listing 4-17. The OnQueryCreated event handler in our code behind for our alternate products page

```
protected void ProdFilter(object sender, QueryCreatedEventArgs e)
{
    var catvalue = (string)Page.RouteData.Values["category"];
    e.Query = from p in e.Query.Cast<Item>()
              where p.ItemCategory.Name == catvalue
              select p;
}
```

The resulting pages look just like the ones in Figures 4-16 and 4-17. The only difference is that we have more control of the filtering and don't need to use a QueryExtender control.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4-8. Building CRUD Operations with an ObjectDataSource Control

Problem

You want to build an ASP.NET page that allows inserting, updating, deleting, and reading from your model using an ObjectDataSource control.

Solution

Suppose you have a model like the one in Figure 4-18.

Licensed by Matt Allen 2175280

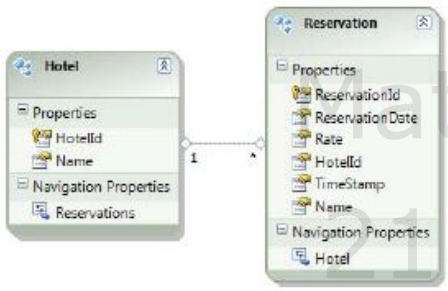


Figure 4-18. *A model for hotel reservations*

Our model represents hotels and their reservations. We want to use an ObjectDataSource control to perform inserts, updates, deletes, and, of course, select operations against this model. To do this, we first need to create a couple of objects that will serve as the sources of our data. We loosely follow the Repository Pattern in Listing 4-18 in building these objects.

Listing 4-18. Our HotelRepository and ReservationRepository classes

```
public class HotelRepository
{
    private EFRecipesEntities context;

    public HotelRepository()
    {
        this.context = new EFRecipesEntities();
    }

    public void Dispose()
    {
        this.context.Dispose();
    }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        public List<Hotel> GetHotels()
        {
            return this.context.Hotels.OrderBy(h => h.Name).ToList();
        }
    }

    public class ReservationRepository
     {
         private EFRecipesEntities context;

         public ReservationRepository()
         {
             this.context = new EFRecipesEntities();
         }

         public void Dispose()
         {
             this.context.Dispose();
         }

         public List<Reservation> GetReservations(string sort,
                                  int startRowIndex, int maximumRows)
         {
             return this.context.Reservations.Include("Hotel")
                  .OrderBy("it." + (sort == string.Empty ? "Name" : sort))
                  .Skip(startRowIndex).Take(maximumRows).ToList();
         }

         public int ReservationCount()
         {
             return this.context.Reservations.Count();
         }

         public void Insert(Reservation reservation)
         {
             this.context.Reservations.AddObject(reservation);
             context.SaveChanges();
         }

         public void Update(Reservation reservation)
         {
             this.context.Reservations.Attach(reservation);
             this.context.ObjectStateManager
                 .ChangeObjectState(reservation, EntityState.Modified);
             this.context.SaveChanges();
         }

         public void Delete(Reservation reservation)
         {
             this.context.Reservations.Attach(reservation);
             this.context.Reservations.DeleteObject(reservation);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        this.context.SaveChanges();
    }
}
```

Once we have the object that will supply our data, we can construct the ASP.NET page that uses the ObjectDataSource control to perform the inserts, update, deletes, and selection of the data. This page is shown in Listing 4-19. The code behind for the page is shown in Listing 4-20.

Listing 4-19. The ASP.NET page using the ObjectDataSource control

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:ListView ID="reservationList" runat="server"
        DataSourceId="reservationSource" DataKeyNames="ReservationId,TimeStamp"
        InsertItemPosition="LastItem">
        <EditItemTemplate>
          <tr>
            <td>
              <asp:Button runat="server" CommandName="Update" Text="Update" />
              <asp:Button runat="server" CommandName="Cancel" Text="Cancel" />
            </td>
            <td>
              <asp:TextBox ID="nameTextBox" runat="server"
                Text='<%# Bind("Name") %>' />
            </td>
            <td>
              <asp:DropDownList ID="hotel" runat="server"
                   AppendDataBoundItems="true"
                   SelectedValue = '<%# Bind("HotelId") %>'
                   DataSourceID="HotelSource" DataTextField="Name"
                   DataValueField="HotelId">
                <asp:ListItem Text="Select" Value="" />
              </asp:DropDownList>
              <asp:ObjectDataSource ID="hotelSource" runat="server"
                   TypeName="Recipe8.HotelRepository"
                   SelectMethod="GetHotels" />
            </td>
            <td>
              <asp:TextBox ID="ResDateTextBox" runat="server"
                   Text='<%# Bind("ReservationDate") %>' />
            </td>
            <td>
              <asp:TextBox ID="RateTextBox" runat="server"
                   Text='<%# Bind("Rate") %>' />
            </td>
          </tr>
        </EditItemTemplate>
        <InsertItemTemplate>
          <tr>
            <td>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        <asp:Button runat="server" CommandName="Insert" Text="Insert" />
        <asp:Button runat="server" CommandName="Cancel" Text="Cancel" />
      </td>
      <td>
        <asp:TextBox ID="nameTextBox" runat="server"
             Text='<%# Bind("Name") %>' />
      </td>
      <td>
        <asp:DropDownList ID="hotel" runat="server"
             AppendDataBoundItems="true"
             SelectedValue='<%# Bind("HotelId") %>'
             DataSourceID="hotelSource"
             DataTextField="Name" DataValueField="HotelId">
          <asp:ListItem Text="Select" Value="" />
        </asp:DropDownList>
        <asp:ObjectDataSource ID="hotelSource" runat="server"
            TypeName="Recipe8.HotelRepository" SelectMethod="GetHotels" />
      </td>
      <td>
        <asp:TextBox ID="ResDateTextBox" runat="server"
             Text='<%# Bind("ReservationDate") %>' />
      </td>
      <td>
        <asp:TextBox ID="RateTextBox" runat="server"
             Text='<%# Bind("Rate") %>' />
      </td>
    </tr>
  </InsertItemTemplate>
  <ItemTemplate>
    <tr>
      <td>
        <asp:Button runat="server" CommandName="Delete" Text="Delete" />
        <asp:Button runat="server" CommandName="Edit" Text="Edit" />
      </td>
      <td><%# Eval("Name") %></td>
      <td><%# Eval("Hotel.Name") %></td>
      <td><%# Eval("ReservationDate") %></td>
      <td><%# Eval("Rate") %></td>
    </tr>
  </ItemTemplate>
  <LayoutTemplate>
    <table>
      <tr>
        <th></th>
        <th>
          <asp:LinkButton runat="server" CommandName="Sort"
               CommandArgument="Name" Text="Name" />
        </th>
        <th>
          <asp:LinkButton runat="server" CommandName="Sort"
               CommandArgument="Hotel.Name" Text="Hotel" />
        </th>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            <th>
              <asp:LinkButton runat="server" CommandName="Sort"
                  CommandArgument="ReservationDate" Text="Reservation Date" />
            </th>
            <th>
              <asp:LinkButton runat="server" CommandName="Sort"
                  CommandArgument="Rate" Text="Daily Rate" />
            </th>
          </tr>
          <tr ID="itemPlaceholder" runat="server" />
        </table>
      </LayoutTemplate>
    </asp:ListView>
    <asp:DataPager ID="pager" runat="server"
         PagedControlID="reservationList" PageSize="2">
      <Fields>
        <asp:NumericPagerField />
      </Fields>
    </asp:DataPager>
    <asp:ObjectDataSource ID="reservationSource" runat="server"
         DataObjectTypeName="Recipe8.Reservation"
        DeleteMethod="Delete" InsertMethod="Insert"
        SelectMethod="GetReservations" UpdateMethod="Update"
        EnablePaging="true" SortParameterName="sort"
        SelectCountMethod="ReservationCount"
        TypeName="Recipe8.ReservationRepository" />
  </div>
 </form>
</body>
```

Listing 4-20. The code behind for the page in Listing 4-19

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            using (var context = new EFRecipesEntities())
            {
                // delete all test data
                context.ExecuteStoreCommand("delete from chapter4.reservation");
                context.ExecuteStoreCommand("delete from chapter4.hotel");

                // insert new test data
                var h1 = new Hotel { Name = "Riverside Inn" };
                var h2 = new Hotel { Name = "Greenville Inn" };
                context.Reservations.AddObject(new Reservation {
                    Name = "Robin Rosen",
                    ReservationDate = DateTime.Parse("4/20/2010"),
                    Rate = 99.95M, Hotel = h1 });
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                context.Reservations.AddObject(new Reservation {
                    Name = "James Marlowe",
                    ReservationDate = DateTime.Parse("5/18/2010"),
                    Rate = 105.00M, Hotel = h2 });
                context.SaveChanges();
            }
        }
    }
}
```

Figure 4-19 shows the ASP.NET page rendered in a browser.

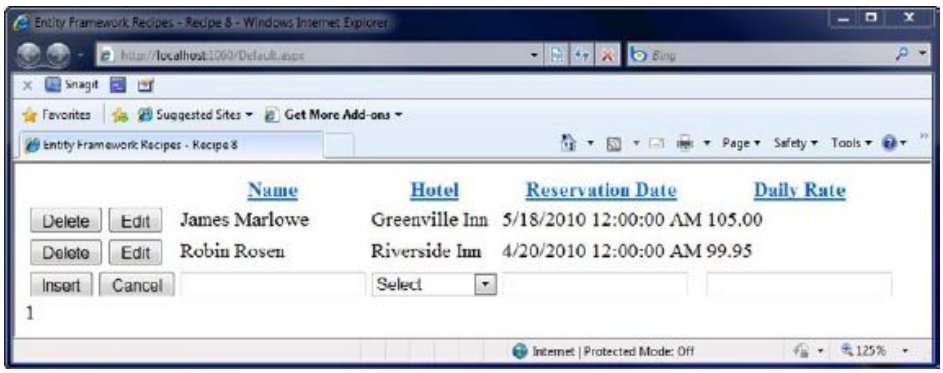


Figure 4-19. *The page supporting inserts, updates, deletes, and listing*

How It Works

Part of the solution is the two classes, HotelRepository and ReservationRepository, which provide the needed operations on the underlying entities. These classes very roughly follow the widely used Repository Pattern. Our ObjectDataSource control uses these two classes to perform the CRUD operations. The bulk of the code in the ASP.NET page is used in the ListView control for InsertTemplate, EditTemplate, and LayoutTemplate.

It is important to note here that the startRowIndex and maximumRows parameters to the ReservationRepository's **`GetReservations()`** method are not arbitrary names. These are the default parameter names used by the ObjectDataSource control for paging. If you need to use different parameter names, these must be specified in the definition of the ObjectDataSource control using the StartRowIndexParameterName and MaximumRowsParameterName attributes. For the Reservation entity we use both the ReservationId and TimeStamp properties to avoid a concurrency violation on updates. We set both ReservationId and TimeStamp as DataKeyNames for the ListView. On post back, the ListView control saves the keys in the control state to preserve the original values. On update, the ObjectDataSource control gets the new values from the **`Bind()`** parameters and the original ReservationId and TimeStamp values from the ListView control.

In the **`Update()`** method of the ReservationRepository, we **`Attach()`** the reservation and then change its status using the **`ChangeObjectStatus()`** method. The drawback to this approach is that we end up marking all the scalar properties except the entity key and the currency column, as modified. All these properties changed or not, will be part of the update statement. Because our model uses foreign key

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

associations, when we change the HotelId property for a reservation, we are also changing the association to the hotel as well.

The **Delete()** method also uses **Attach()** and **ChangeObjectState()** to change the state of the object to Deleted. For this, we need only the entity key.

4-9. Using Entity Framework With MVC

Problem

You want to use Entity Framework with ASP.NET MVC.

Solution

Suppose you have a table in your database like the one shown in Figure 4-20.

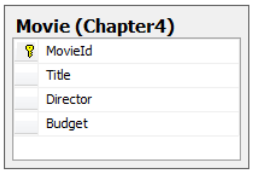


Figure 4-20. *A table with some information about a movie*

This table holds some information about movies. You want to create a simple ASP.NET MVC web application that uses Entity Framework. To create a web application that provides for inserting, updating, and deleting movies in our Movie table, do the following:

1. Add a new ASP.NET MVC 2 Empty Web Application to your solution.
2. Right-click the Models folder and select Add ➤ New Item. Add an ADO.NET Entity Data Model. Import the movie table in Figure 4-20. The model should look like the one in Figure 4-21.
3. Right-click the Controllers folder and select Add ➤ Controller. Name the new controller **HomeController**. Check the box: Add action methods for Create, Update, and Details scenarios. Click Add.
4. Add a private variable to the HomeController to hold the context:

```
private EFRecipesEntities context = new EFRecipesEntities();
```

5. Change the **Index()** method in the HomeController to return the list of movies. Use the following code:

```
public ActionResult Index()
 {
    return View(context.Movies.ToList());
 }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

6. Right-click the **Index()** method and select Add ➤ View. Uncheck Select master page. Select Movie for the view data class and List as the content. See Figure 4-22.

7. Change the **Details()** method in the HomeController to return a single movie based on the MovieId. Use the following code:

```
public ActionResult Details(int id)
 {
    var movie = context.Movies.Single(m => m.MovieId == id);
    return View(movie);
 }
```

8. Right-click the **Details()** method and select Add ➤ View. Set the view data class to Movie and the view content to Details. See Figure 4-23.

9. Right-click the **Create()** method and select Add ➤ View. Set the view data class to Movie and the view content to Create. See Figure 4-24.

10. Use the code in Listing 4-21 to replace the code for the overloaded **Create()** method that takes a FormCollection. This is the method that is decorated with the HttpPost attribute.

11. Change the **Edit()** method in the HomeController to return a single movie based on the id. Use the following code:

```
public ActionResult Edit(int id)
 {
    var movie = context.Movies.Single(m => m.MovieId == id);
    return View(movie);
 }
```

12. Right-click the **Edit()** method and select Add ➤ View. Set the view data class to Movie and view content to Edit. See Figure 4-25.

13. The second **Edit()** method that takes a FormsCollection is called when the user has modified the movie's properties and has submitted the form to the server. Use the code in Listing 4-22 to replace the code for this second **Edit()** method.

14. We need to add a Delete button to the Index.aspx page in the Views ➤ Home folder. Use the code in Listing 4-23 to add this button next to the Edit link.

15. Add a **Delete()** method to handle the Delete button click. Use the code in Listing 4-24 to implement the **Delete()** method in HomeController.cs file.

16. Run the web application and add a few movies. After you have added a few movies, the web application should look something like the one shown in Figure 4-26.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

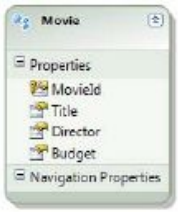


Figure 4-21. *The model created from the Movie table*

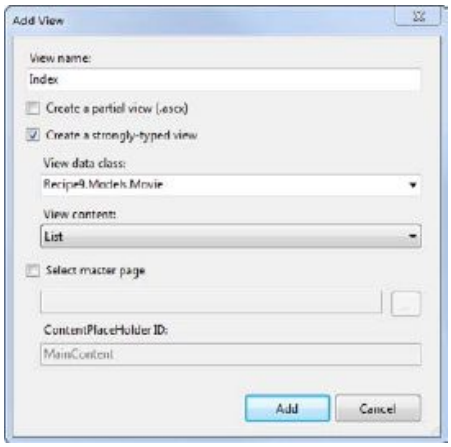


Figure 4-22. *Adding a view for the* ***`Index()`*** *method*

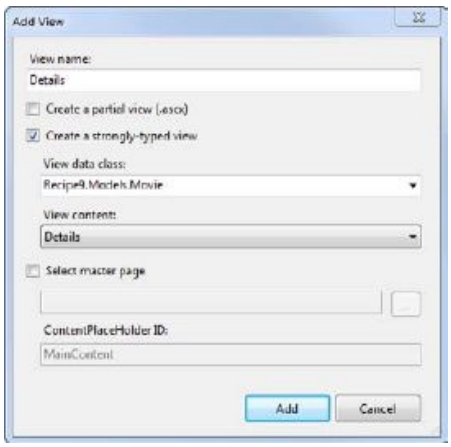


Figure 4-23. *Adding a view for the* ***`Details()`*** *method*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

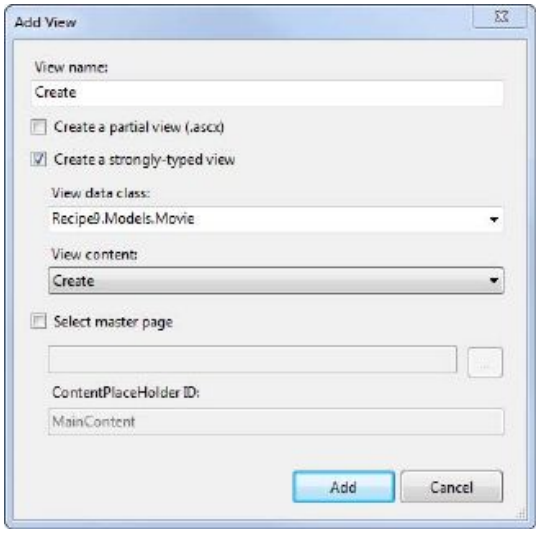


Figure 4-24. *Adding a view for the* ***`Create()`*** *method*

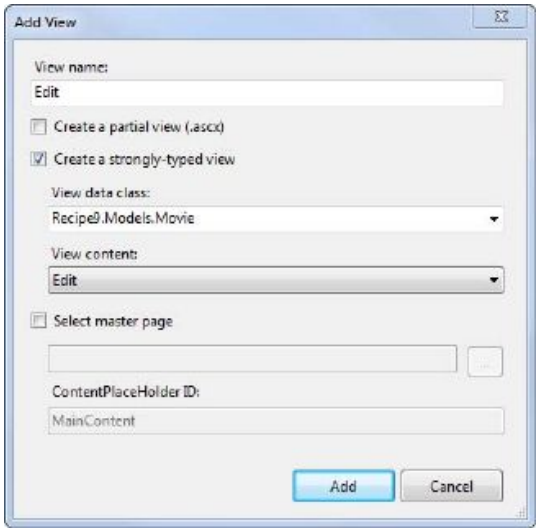


Figure 4-25. *Adding a view for the* ***`Edit()`*** *method*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

*Listing 4-21. The code for the second **Create()** method that handles posts*

```
[HttpPost]
public ActionResult Create([Bind(Exclude = "MovieId")] Movie movie)
{
    try
    {
        context.Movies.AddObject(movie);
        context.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

Licensed by Matt Allen 2175280

*Listing 4-22. The code for the second **Edit()** method that handles posts*

```
[HttpPost]
public ActionResult Edit(int id, Movie movie)
{
    try
    {
        movie.MovieId = id;
        context.Movies.Attach(movie);
        context.ObjectStateManager.ChangeObjectState(movie,
                                                     EntityState.Modified);
        context.SaveChanges();
        return RedirectToAction("Index");
    }
    catch
    {
        return View();
    }
}
```

Listing 4-23. Adding the Delete button to the Index.aspx page

```
<td>
    <% using (Html.BeginForm("Delete", "Home", new { id = item.MovieId }))
       { %> <input type="submit" value="Delete" /> <% } %>
</td>
```

*Listing 4-24. The **Delete()** method that handles the Delete button click*

```
[AcceptVerbs(HttpVerbs.Post)]
public ActionResult Delete(int id)
{
    try
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            var movie = new Movie { MovieId = id };
            context.Movies.Attach(movie);
            context.Movies.DeleteObject(movie);
            context.SaveChanges();

            return RedirectToAction("Index");
        }
        catch
        {
            return View();
        }
}
```

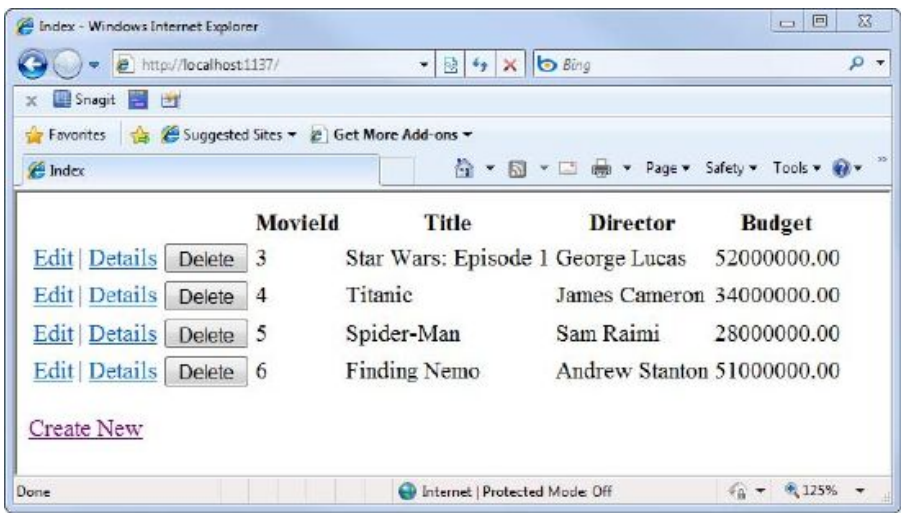


Figure 4-26. *The rendered Index.aspx page with a few movies from our database*

How It Works

We created a very simple ASP.NET MVC web application that uses the Entity Framework model in Figure 4-21 that we created by importing the database table in Figure 4-20. The web application lists the current movies and supports editing, creating, and deleting movies from the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 5

Loading Entities and Navigation Properties

Entity Framework provides a rich modeling environment representing a conceptual view of the underlying objects and relationships in data storage. The recipes in this chapter show you how to control the loading of instances of related entities in your queries.

The default behavior for Entity Framework is to load only the entities directly accessed by your application. In general, this is exactly what you want. If Entity Framework aggressively loaded all the entities related through one or more associations, you would likely end up loading more entities than you needed. This would increase the memory footprint of your application and slow it down.

In Entity Framework, you can control when the loading of related entities occurs and optimize the number of database queries executed. Carefully managing when related entities are loaded can increase performance and simplify your code.

We start off this chapter with a number of recipes illustrating how to load some or all of the related entities in a single query. This type of loading, also called *eager loading*, is used to both reduce the number of round trips to the database and to more precisely control which related entities are loaded.

Sometimes you need to defer loading of certain related entities because they may be expensive to load or are not used very often. We'll cover a number of scenarios using the `Load()` method to precisely control when to load one or more related entities.

5-1. Loading Related Entities

Problem

You want to load an entity along with some related entities in a single round trip to the database.

Solution

Let's say you have a model like the one shown in Figure 5-1.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

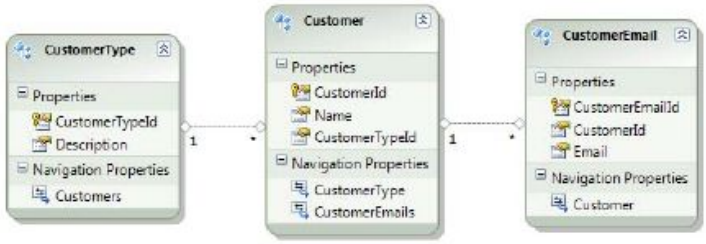


Figure 5-1. *A model with a Customer and its related information*

In this model, we have a Customer entity with a single CustomerType and perhaps many CustomerEmail addresses. The association with CustomerType is one-to-many with CustomerType on the one side of the association. This is an entity reference.

The association with CustomerEmail is also one-to-many but with CustomerEmail on the many side of the association. This is an entity collection.

To include all the instances of the related CustomerEmail entity as well as the instance of the related CustomerType entity when retrieving instances of the Customer entity, use the **`Include()`** method syntax as shown in Listing 5-1.

Listing 5-1. Eager loading of instances of Customertype and Customeremail along with instances of Customer

```
using (var context = new EFRecipesEntities())
{
    var web = new CustomerType { Description = "Web Customer",
                                 CustomerTypeId = 1 };
    var retail = new CustomerType { Description = "Retail Customer",
                                    CustomerTypeId = 2 };
    var customer = new Customer { Name = "Joan Smith", CustomerType = web };
    customer.CustomerEmails.Add(new CustomerEmail
                                     { Email = "jsmith@gmail.com" });
    customer.CustomerEmails.Add(new CustomerEmail { Email = "joan@smith.com" });
    context.Customers.AddObject(customer);
    customer = new Customer { Name = "Bill Meyers", CustomerType = retail };
    customer.CustomerEmails.Add(new CustomerEmail
                                     { Email = "bmeyers@gmail.com" });
    context.Customers.AddObject(customer);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var customers = context.Customers.Include("CustomerType")
                                     .Include("CustomerEmails");
    Console.WriteLine("Customers");
    Console.WriteLine("=========");
    foreach (var customer in customers)
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        Console.WriteLine("{0} is a {1}, email address(es)", customer.Name,
                          customer.CustomerType.Description);
        foreach (var email in customer.CustomerEmails)
        {
            Console.WriteLine("\t{0}", email.Email);
        }
    }
}

using (var context = new EFRecipesEntities())
{
    var customTypes = context.CustomerTypes.Include("Customers.CustomerEmails");
    Console.WriteLine("\nCustomers by Type");
    Console.WriteLine("=================");
    foreach (var customerType in customTypes)
    {
        Console.WriteLine("Customer type: {0}", customerType.Description);
        foreach (var customer in customerType.Customers)
        {
            Console.WriteLine("{0}", customer.Name);
            foreach (var email in customer.CustomerEmails)
            {
                Console.WriteLine("\t{0}", email.Email);
            }
        }
    }
}
```

The output of the code in Listing 5-1 is the following:

```
Customers

=========

Joan Smith is a Web Customer, email address(es)

        jsmith@gmail.com

        joan@smith.com

Bill Meyers is a Retail Customer, email address(es)

        bmeyers@gmail.com



Customers by Type

=================
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Customer type: Web Customer

Joan Smith

        jsmith@gmail.com

        joan@smith.com

Customer type: Retail Customer

Bill Meyers

        bmeyers@gmail.com
```

How It Works

By default, Entity Framework loads only entities that you specifically request. This is an important principle to keep in mind. The alternative, always loading every associated entity, may cause a much larger part of the object graph to be loaded into memory than you might need.

In this example, we used the **Include()** method to eagerly load the related parts of the object graph. **Include()** takes a string representation of the part of the object graph you want to load. This string representation of the partial object graph is also called a path and is made of the navigation property names separated by the '.' character.

In Listing 5-1, we create a couple of instances of the CustomerType entity and use these together with instances of the CustomerEmail entity to create a couple of Customers.

To get the object graph from the Customer, we use the **Include()** method twice. In the first use, we include the entity reference to the CustomerType entity. This is on the one side of the one-to-many association. In the second use of **Include()**, we get the many side of the one-to-many association bringing along all the instances of the CustomerEmail entity for the customer. By using the **Include()** method twice, we pull in all the referenced entities from both of the Customer's navigation properties.

To get the object graph from the CustomerType, we invoke **Include()** just once, but this time we pass in a path that includes both the Customers and the CustomerEmails.

Both of these queries resulted in a load of the entire object graph into the object context. In our example, this wasn't much, but for databases with thousands or millions of customers, we could end up using lots of memory if we're not careful.

The **Include()** method has some important performance implications. On the one hand, loading a large part of the object graph into the object context (that is, into memory), can end up using a lot of memory. If we had millions of customers, this would definitely be a problem. On the other hand, the **Include()** method loads the object graph in one trip to the database. If your application will end up loading each entity in the graph separately, requiring lots of trips to the database, you may end up with a lot less database traffic with the use of the **Include()** method.

There is one more performance issue. To get everything in one trip to the database, Entity Framework might construct a rather unwieldy SQL statement with lots of joins. The SQL statement from our first query is shown in Listing 5-2. Not only is the query getting a little complicated but it's also bringing back duplicate information, as shown in Figure 5-2. When materializing the object graph, Entity Framework has to remove the duplicate information.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The bottom line is that **Include()**, used carefully, can improve performance over piecemeal loading of the entities. Keep in mind the extra memory footprint and the extra work done at the database layer and in Entity Framework.

*Listing 5-2. The SQL query resulting from our use of the **Include()** method*

```
SELECT
[Project1].[CustomerId] AS [CustomerId],
[Project1].[Name] AS [Name],
[Project1].[CustomerTypeId] AS [CustomerTypeId],
[Project1].[CustomerTypeId1] AS [CustomerTypeId1],
[Project1].[Description] AS [Description],
[Project1].[C1] AS [C1],
[Project1].[CustomerEmailId] AS [CustomerEmailId],
[Project1].[CustomerId1] AS [CustomerId1],
[Project1].[Email] AS [Email]
FROM ( SELECT
        [Extent1].[CustomerId] AS [CustomerId],
        [Extent1].[Name] AS [Name],
        [Extent1].[CustomerTypeId] AS [CustomerTypeId],
        [Extent2].[CustomerTypeId] AS [CustomerTypeId1],
        [Extent2].[Description] AS [Description],
        [Extent3].[CustomerEmailId] AS [CustomerEmailId],
        [Extent3].[CustomerId] AS [CustomerId1],
        [Extent3].[Email] AS [Email],
        CASE WHEN ([Extent3].[CustomerEmailId] IS NULL) THEN CAST(NULL AS int) ELSE 1 END AS
[C1]
        FROM   [Chapter5].[Customer] AS [Extent1]
        LEFT OUTER JOIN [Chapter5].[CustomerType] AS [Extent2] ON [Extent1].[CustomerTypeId] =
[Extent2].[CustomerTypeId]
        LEFT OUTER JOIN [Chapter5].[CustomerEmail] AS [Extent3] ON [Extent1].[CustomerId] =
[Extent3].[CustomerId]
)  AS [Project1]
ORDER BY [Project1].[CustomerId] ASC, [Project1].[CustomerTypeId1] ASC, [Project1].[C1] ASC
```

Results | Messages

	CustomerId	Name	CustomerTypeId	CustomerTypeId1	Description	C1	CustomerEmailId	CustomerId1	Email
1	34	Joan Smith	1	1	Web Customer	1	51	34	jsmith@gmail.com
2	34	Joan Smith	1	1	Web Customer	1	52	34	joan@smith.com
3	35	Bill Meyers	2	2	Retail Customer	1	53	35	bmeyers@gmail.com

Figure 5-2. *Redundant data resulting from the Include() method*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5-2. Loading a Complete Object Graph

Problem

You have a model with several related entities and you want to load the complete object graph of all the instances of each entity in a single query.

Solution

Suppose you have a conceptual model like the one in Figure 5-3. Each course has several sections. Each section is taught by an instructor and has several students.

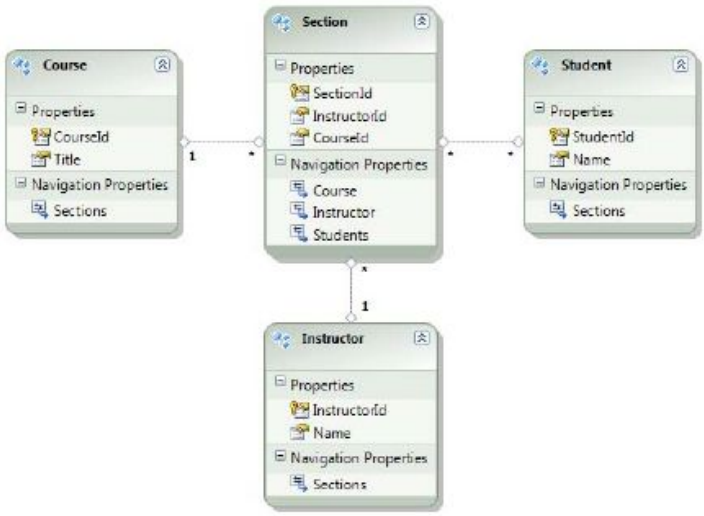


Figure 5-3. *A model with a few related entities*

To retrieve all the courses, sections, instructors, and students represented in the database in a single query, use the **`Include()`** method with a query path parameter, as shown in Listing 5-3.

Listing 5-3. Retrieving an entire object graph in a single query

```
using (var context = new EFRecipesEntities())
{
    var course = new Course { Title = "Biology 101" };
    var fred = new Instructor { Name = "Fred Jones" };
    var julia = new Instructor { Name = "Julia Canfield" };
    var section1 = new Section { Course = course, Instructor = fred };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    var section2 = new Section { Course = course, Instructor = julia };
    var jim = new Student { Name = "Jim Roberts" };
    jim.Sections.Add(section1);
    var jerry = new Student { Name = "Jerry Jones" };
    jerry.Sections.Add(section2);
    var susan = new Student { Name = "Susan O'Reilly" };
    susan.Sections.Add(section1);
    var cathy = new Student { Name = "Cathy Ryan" };
    cathy.Sections.Add(section2);
    context.Courses.AddObject(course);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var graph = context.Courses
                        .Include("Sections.Instructor")
                        .Include("Sections.Students");
    Console.WriteLine("Courses");
    Console.WriteLine("=======");
    foreach (var course in graph)
    {
        Console.WriteLine("{0}", course.Title);
        foreach (var section in course.Sections)
        {
            Console.WriteLine("\tSection: {0}, Instrutor: {1}",
                              section.SectionId.ToString(),
                              section.Instructor.Name);
            Console.WriteLine("\tStudents:");
            foreach (var student in section.Students)
            {
                Console.WriteLine("\t\t{0}", student.Name);
            }
            Console.WriteLine("\n");
        }
    }
}
```

The code in Listing 5-3 produces the following output:

```
Courses

=======

Biology 101

        Section: 7, Instructor: Fred Jones

        Students:
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            Susan O'Reilly

            Jim Roberts


        Section: 8, Instructor: Julia Canfield

        Students:

            Cathy Ryan

            Jerry Jones
```

How It Works

A query path is a string parameter to the **Include()** method. A query path represents the entire path of the object graph that is loaded by the **Include()** method. The **Include()** method extends the query to include the entities referenced along the query path.

In Listing 5-3, we use the **Include()** method twice. **Include()** is invoked first with a query path parameter that includes the part of the graph extending through Section to Instructor. This modifies the query to include all the Sections and their Instructors. The second invocation includes a path extending through Section to Student. This modifies the query to include Sections and their Students. The result is a materialization of the complete object graph including all the Course entities and the entities on each end of all associations in the model.

You can use query paths that use navigation properties to any depth. This gives you a lot of flexibility in partial or complete object graph loading. Entity Framework attempts to optimize the final query generation by pruning off overlapping or duplicate query paths.

The syntax and semantics of the **Include()** method is deceptively simple. Don't let the simplicity fool you into thinking that there is no performance price to be paid when using the **Include()** method. Eager loading with several **Include()** method invocations can rapidly increase the complexity of the query sent to the database and dramatically increase the amount of data returned from the database. The complex queries generated can lead to poor performance plan generation and the large amount of returned data can cause Entity Framework to spend an inordinate amount of time removing duplicate data.

5-3. Loading Navigation Properties on Derived Types

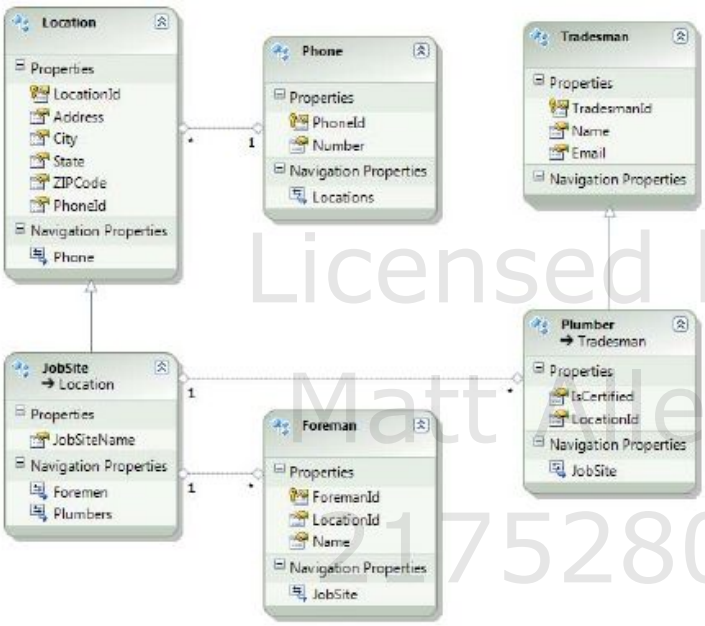
Problem

You have a model with one or more derived types that are in a has-a relationship with one or more other entities. You want to eagerly load all the related entities in one round trip to the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Suppose you have a model like the one in Figure 5-4.



Licensed by Matt Allen 2175280

Figure 5-4. *A model for Plumbers with their JobSite and other related entities*

In this model, the Plumber entity extends the Tradesman entity. A Plumber has a JobSite that is represented by a one-to-many association. The JobSite type extends the Location entity. Location has a Phone, which is represented by a one-to-many association. Finally, a JobSite can have zero or more Foremen. This is also represented by a one-to-many association.

Suppose you want to retrieve a plumber, the jobsite she works on, the jobsite's phone number, and all the foremen at the jobsite. You want to retrieve all this in one round trip to the database.

The code in Listing 5-4 illustrates one way to use the **`Include()`** method to eagerly load the related entities in one query.

Listing 5-4. Retrieving related entities in one round trip to the database using eager loading with the Include() method

```
using (var context = new EFRecipesEntities())
{
    var foreman1 = new Foreman { Name = "Carl Ramsey" };
    var foreman2 = new Foreman { Name = "Nancy Ortega" };
    var phone = new Phone { Number = "817 867-5309" };
    var jobsite = new JobSite { JobSiteName = "City Arena",
                                Address = "123 Main", City = "Anytown",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                State = "TX", ZIPCode = "76082",
                                Phone = phone };
    jobsite.Foremen.Add(foreman1);
    jobsite.Foremen.Add(foreman2);
    var plumber = new Plumber { Name = "Jill Nichols",
                                Email = "JNichols@plumbers.com",
                                JobSite = jobsite };
    context.Tradesmen.AddObject(plumber);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var plumber = context.Tradesmen.OfType<Plumber>()
                                   .Include("JobSite.Phone")
                                   .Include("JobSite.Foremen").First();
    Console.WriteLine("Plumber's Name: {0} ({1})", plumber.Name, plumber.Email);
    Console.WriteLine("Job Site: {0}", plumber.JobSite.JobSiteName);
    Console.WriteLine("Job Site Phone: {0}", plumber.JobSite.Phone.Number);
    Console.WriteLine("Job Site Foremen:");
    foreach (var boss in plumber.JobSite.Foremen)
    {
        Console.WriteLine("\t{0}", boss.Name);
    }
}
```

The following output is produced by code in Listing 5-4:

```
Plumber's Name: Jill Nichols (JNichols@plumbers.com)

Job Site: City Arena

Job Site Phone: 817 867-5309

Job Site Foremen:

        Carl Ramsey

        Nancy Ortega
```

How It Works

Our query starts by selecting instances of the derived type Plumber. To get these, we use the `OfType<Plumber>()` method. The `OfType<>()` method select instances of the given subtype from the entity set.

From Plumber, we want to load the related JobSite and the Phone for the JobSite. Notice that the JobSite entity does not have a Phone navigation property, but JobSite derives from Location, which does

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

have a Phone navigation property. Because Phone is a property of the base entity, it's also available on the derived entity. That's the beauty of inheritance. This makes the query path simply: "JobSite.Phone".

We used the **Include()** method again with a query path that references the Foreman entities from the JobSite entity. Here we have a one-to-many association JobSite and Foreman. Notice the navigation property was pluralized by the wizard (from Foreman to Foremen).

Finally, we use the **First()** method to select just the first Plumber instance.

The resulting query is somewhat complex, involving several joins and subselects. The alternative, using the **Load()** method for each related entity, would require several round trips to the database and would result in a performance hit, especially if we retrieved many Plumbers.

5-4. Using Include() with Other LINQ Query Operators

Problem

You have a LINQ query that uses operators such as **group by**, **join**, and **where**; and you want to use the **Include()** method to eagerly load additional entities.

Solution

Let's say you have a model like the one shown in Figure 5-5.

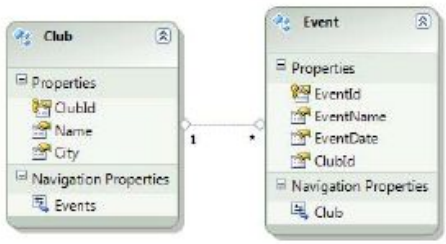


Figure 5-5. *A simple model with a one-to-many association between Club and Event*

To use the **Include()** method in combination with a **group by** clause, form the LINQ expression without the **Include()** method first; then cast the expression as an **ObjectQuery<T>** and invoke the **Include()** method. The code in Listing 5-5 demonstrates this approach.

Listing 5-5. Casting to ***ObjectQuery<T>*** *and Invoking* ***Include()***

```
using (var context = new EFRecipesEntities())
{
    var club = new Club { Name = "Star City Chess Club", City = "New York" };
    context.Clubs.AddObject(club);
    new Event { EventName = "Mid Cities Tournament",
                EventDate = DateTime.Parse("1/09/2010"), Club = club };
    new Event { EventName = "State Finals Tournament",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                EventDate = DateTime.Parse("2/12/2010"), Club = club };
    new Event { EventName = "Winter Classic",
                EventDate = DateTime.Parse("12/18/2009"), Club = club };
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var events = from ev in context.Events
                 where ev.Club.City == "New York"
                 group ev by ev.Club into g
                 select g.FirstOrDefault(e1 =>
                            e1.EventDate == g.Min(evt => evt.EventDate));
    var e = ((ObjectQuery<Event>)events).Include("Club").First();
    Console.WriteLine("The next New York club event is:");
    Console.WriteLine("\tEvent: {0}", e.EventName);
    Console.WriteLine("\tDate: {0}", e.EventDate.ToShortDateString());
    Console.WriteLine("\tClub: {0}", e.Club.Name);
}
```

The output of the code in Listing 5-5 is the following:

```
The next New York club event is:

        Event: Winter Classic

        Date: 12/18/2009

        Club: Star City Chess Club
```

How It Works

We start by creating a Club and three Events. The code looks a little strange. We created the events without assigning them to anything! Well, not quite. The initializer for each assigns the Club property the instance of the club we created at the top. This is all that is needed to add the events to the club's event entity collection. There is no reason to keep another set of references to the events. They are already referenced by the club.

In the query, we grab all the events at clubs in New York, group them by club, and find the first one in date order. The events variable holds just the expression. It hasn't executed anything on the database yet.

Next, we cast the expression to `ObjectQuery<Event>`. This is required because the LINQ expressions are of type `IQueryable<T>`. Because `ObjectQuery<T>` implements `IQueryable<T>`,and our LINQ to Entities expression is really of type `ObjectQuery<T>`, it's safe to do the cast. But why do we need to cast it? Because `IQueryable<T>` doesn't have an `Include()` method, but `ObjectQuery<T>` does have it. The cast gives us access to the `Include()` method.

Many developers find the `Include()` method a little confusing. In some cases, Intellisense will not show it as available (because of the type of the expression). In some cases, it will be silently ignored at runtime. Surprisingly, the compiler rarely complains unless it cannot determine the resulting type. The

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

problems usually show up at runtime when they can be a more difficult fix. Here are some simple rules to follow when using **Include()**:

1. Include() applies only to the final query results. When Include() is applied to a subquery, join, or nested from clause, it is ignored when the command tree is generated.
2. The Include() method is an extension method on type ObjectQuery<T>. If the expression is of type IQueryable<T>, it must be cast to ObjectQuery<T> before the Include() method will be available.
3. Include() can be applied only to results that are entities. If the expression projects results that are not entities, Include() will be ignored.
4. The query cannot change the type of the results between the Include() and the outermost operation. A group by clause, for example, changes the type of the results.
5. The query path used in the Include() expression must start at a navigation property on the type returned from the outermost operation. The query path cannot start at an arbitrary point.

Let's see how these rules apply to the code in Listing 5-5.

The query groups the events by the sponsoring club. The **group by** operator changes the result type from **Event** to a grouping result. Here Rule 4 says that we need to invoke the **Include()** method after the **group by** clause has changed the type. We do this by invoking **Include()** at the very end. If we applied the **Include()** method earlier as in **from ev in context.Events.Include()**, the **Include()** method would have been silently dropped from the command tree and never applied.

If you mouse over the events variable, you will notice that Intellisense shows the type as **IQueryable<Event>**. Rule 2 says that **IQueryable<T>** does not implement the **Include()** method. However, we know that **events** is really of type **ObjectQuery<Event>**, so following Rule 2, we cast **events** to **ObjectQuery<Event>** and then invoke the **Include()** method.

5-5. Deferred Loading of Related Entities

Problem

You have an instance of an entity and you want to defer load two or more related entities in a single query.

Solution

Suppose you have a model like the one in Figure 5-6.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

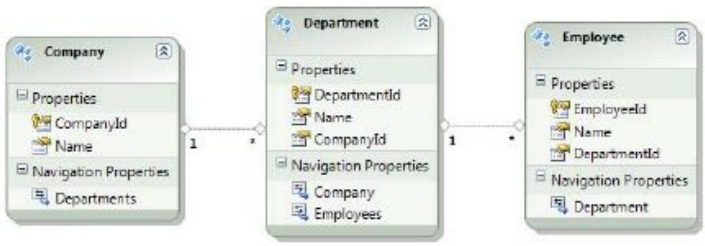


Figure 5-6. *A a model with an employee, her department, and the department's company*

In the model shown in Figure 5-6, an Employee is associated with exactly one Department. Each Department is associated with exactly one Company.

Given an instance of an Employee, you want to load both his department and the department's company. What makes this problem somewhat unique is that we already have an instance of Employee and we want to avoid going back to the database to get another copy of the Employee just so that we can use the **Include()** method to obtain the related instances of Company and Department. Perhaps in your real-world problem, Employee is a very expensive entity to retrieve and materialize.

We could use the **Load()** method twice to load the related Department instance and then again to load the related Company instance. However, this would generate two round trips to the database. To load the related instances using just one query, we can either re-query the Employee entity set using the **Include()** method with a query path including the Department and the Company, or use the **CreateSourceQuery()** method on the DepartmentReference property. The code in Listing 5-6 shows both approaches.

Listing 5-6. Inserting into the model and retrieving the related entities using two slightly different approaches

```
using (var context = new EFRecipesEntities())
{
    var company = new Company { Name = "Acme Products" };
    var acc = new Department { Name = "Accounting", Company = company };
    var ship = new Department { Name = "Shipping", Company = company };
    var emp1 = new Employee { Name = "Jill Carpenter", Department = acc };
    var emp2 = new Employee { Name = "Steven Hill", Department = ship };
    context.Employees.AddObject(emp1);
    context.Employees.AddObject(emp2);
    context.SaveChanges();
}

// first approach
using (var context = new EFRecipesEntities())
{
    // assume we already have an employee
    var jill = context.Employees.Where(o => o.Name == "Jill Carpenter").First();

    // now get Jill's department and company
    var results = context.Employees.Include("Department.Company")
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                    .Where(o => o.EmployeeId == jill.EmployeeId).First<Employee>();
    Console.WriteLine("{0} works in {1} for {2}", jill.Name,
                        jill.Department.Name, jill.Department.Company.Name);
}

// more efficient, does not retrieve employee again
using (var context = new EFRecipesEntities())
{
    // assume we already have an employee
    var jill = context.Employees.Where(o => o.Name == "Jill Carpenter").First();

    var moreResults = jill.DepartmentReference.CreateSourceQuery()
                                            .Include("Company").First();
    context.Attach(moreResults);
    Console.WriteLine("{0} works in {1} for {2}", jill.Name,
                        jill.Department.Name, jill.Department.Company.Name);
}
```

The following is the output of the code in Listing 5-6:

```
Jill Carpenter works in Accounting for Acme Products

Jill Carpenter works in Accounting for Acme Products
```

How It Works

If we didn't already have an instance the Employee entity, we could simply use the **Include()** method with a query path "Department.Company". This is essentially the approach we take in the first query. The disadvantage of this approach is that it retrieves all the columns for the Employee entity. In many cases, this might be an expensive operation. Because we already have this object in the context, it seems wasteful to gather these columns again from the database and transmit them across the wire.

In the second query, we use the **CreateSourceQuery()** method available on the DepartmentReference property to retrieve the related instance of the Department entity as well as the instance of the Company entity. This second approach is more efficient because it does not retrieve the Employee columns. Our use of the **Attach()** method to attach the retrieved Department instance to the object context is not strictly required in this case because of relationship span.

5-6. Filtering and Ordering Related Entities

Problem

You have an instance of an entity and you want to load a related EntityCollection applying both a filter and an ordering.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Suppose you have a model like the one shown in Figure 5-7.

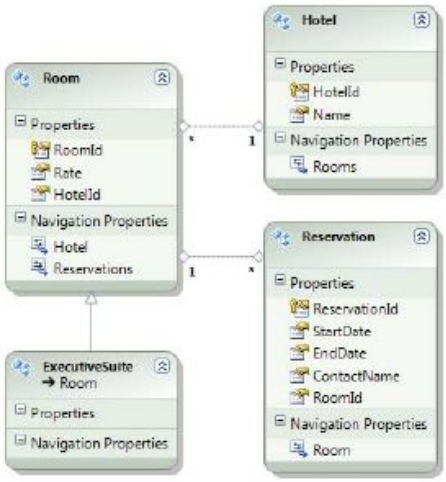


Figure 5-7. *A model for a hotel reservation system*

Let's assume we have an instance of a Hotel entity. To retrieve the executive suite rooms for the hotel, see which have reservations, and order them by room rate, use the pattern shown in Listing 5-7.

Listing 5-7. Filtering and ordering an entity collection using CreateSourceQuery()

```
using (var context = new EFRecipesEntities())
{
    var hotel = new Hotel { Name = "Grand Seasons Hotel" };
    var r101 = new Room { Rate = 79.95M, Hotel = hotel };
    var es201 = new ExecutiveSuite { Rate = 179.95M, Hotel = hotel };
    var es301 = new ExecutiveSuite { Rate = 299.95M, Hotel = hotel };
    var res1 = new Reservation { StartDate = DateTime.Parse("3/12/2010"),
                                 EndDate = DateTime.Parse("3/14/2010"),
                                 ContactName = "Roberta Jones", Room = es301 };
    var res2 = new Reservation { StartDate = DateTime.Parse("1/18/2010"),
                                 EndDate = DateTime.Parse("1/28/2010"),
                                 ContactName = "Bill Meyers", Room = es301 };
    var res3 = new Reservation { StartDate = DateTime.Parse("2/5/2010"),
                                 EndDate = DateTime.Parse("2/6/2010"),
                                 ContactName = "Robin Rosen", Room = r101 };
    context.Hotels.AddObject(hotel);
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    // assume we have an instance of hotel
    var hotel = context.Hotels.First();

    var rooms = hotel.Rooms.CreateSourceQuery()
                           .Include("Reservations")
                           .Where(r => r is ExecutiveSuite &&
                                      r.Reservations.Any())
                           .OrderBy(r => r.Rate);
    Console.WriteLine("Executive Suites for {0} with reservations", hotel.Name);
    hotel.Rooms.Attach(rooms);
    foreach (var room in hotel.Rooms)
    {
        Console.WriteLine("\nExecutive Suite {0} is {1} per night",
                          room.RoomId.ToString(), room.Rate.ToString("C"));
        Console.WriteLine("Current reservations are:");
        foreach (var res in room.Reservations.OrderBy(r => r.StartDate))
        {
            Console.WriteLine("\t{0} thru {1} ({2})",
                              res.StartDate.ToShortDateString(),
                              res.EndDate.ToShortDateString(),
                              res.ContactName);
        }
    }
}
```

The following is the output of the code shown in Listing 5-7:

```
Executive Suites for Grand Seasons Hotel with reservations



Executive Suite 84 is $299.95 per night

Current reservations are:

        1/18/2010 thru 1/28/2010 (Bill Meyers)

        3/12/2010 thru 3/14/2010 (Roberta Jones)
```

How It Works

The code in Listing 5-7 uses the **CreateSourceQuery()** method to get access to the query that is used to retrieve the entity collection on the navigation property. We apply the **Include()** method to eagerly load the associated reservations for each room. We apply **Include()** before the **where** clause because prior to the **where** clause, the expression is of type **ObjectQuery<Room>**, which exposes the **Include()** method.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

After the **where** clause, the expression is of type **IQueryable<Room>**, which does not have the **Include()** method.

The **where** clause filters the collection to rooms of type ExecutiveSuite that have at least one reservation. We then order the collection by room rate using an **OrderBy** clause.

After we obtain the filtered and ordered collection of rooms with their reservations, we use the **Attach()** method to connect the collection to the instance of the Hotel entity. Once attached, we iterate through the rooms. For each room, we order the reservations for the room by start date. This second ordering is done in memory on the entity collection while the first ordering and filtering was performed in the database.

One way to simplify the filter is to use the **OfType<T>()** method, as shown in the code snippet in Listing 5-8. This approach relies on .NET 4.0's new support for covariance and contravariance. Now the type of rooms passed to the **Attach()** method is **IOrderedQueryable<ExecutiveSuite>**, which defines methods whose signatures reference the ExecutiveSuite type that is derived from the Room entity.

*Listing 5-8. Using **OfType<T>** to filter by derived type*

```
var rooms = hotel.Rooms
                 .CreateSourceQuery()
                 .Include("Reservations")
                 .OfType<ExecutiveSuite>()
                 .Where(r => r.Reservations.Any()).OrderBy(r => r.Rate);
hotel.Rooms.Attach(rooms);
```

5-7. Executing Aggregate Operations on Related Entities

Problem

You want to apply an aggregate operator on a related entity collection without loading the collection.

Solution

Suppose you have a model like the one shown in Figure 5-8.

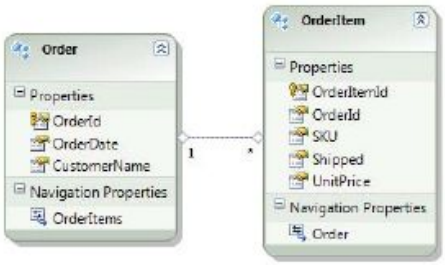


Figure 5-8. *Orders and their associated order items*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In Figure 5-8, we have a simple model composed of an order and the products (collection of OrderItems) shipped for the order. One way to get the total amount for the order is to use the **Load()** method to load the EntityCollection of order items and then iterate through this collection, calculating the sum of the amount for each order item.

Another way to get the same result is to push the iteration to the data store layer, letting it compute the total amount. The advantage to this second approach is that we avoid the potentially costly overhead of materializing each order item for the sole purpose of summing the total order amount. To implement this second approach, follow the pattern shown in Listing 5-9.

Listing 5-9. Applying an aggregate operator on related entities without loading them

```
using (var context = new EFRecipesEntities())
{
    var order = new Order { CustomerName = "Jenny Craig",
                            OrderDate = DateTime.Parse("3/12/2010") };
    var item1 = new OrderItem { Order = order, Shipped = 3, SKU = 2827,
                                UnitPrice = 12.95M };
    var item2 = new OrderItem { Order = order, Shipped = 1, SKU = 1918,
                                UnitPrice = 19.95M };
    var item3 = new OrderItem { Order = order, Shipped = 3, SKU = 392,
                                UnitPrice = 8.95M };
    context.Orders.AddObject(order);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // assume we have an instance of Order
    var order = context.Orders.First();

    // get the total order amount
    var amt = order.OrderItems.CreateSourceQuery()
                              .Sum(o => (o.Shipped * o.UnitPrice));
    Console.WriteLine("Order Number: {0}", order.OrderId.ToString());
    Console.WriteLine("Order Date: {0}", order.OrderDate.ToShortDateString());
    Console.WriteLine("Order Total: {0}", amt.ToString("C"));
}
```

The following is the output of the code in Listing 5-9:

```
Order Number: 6

Order Date: 3/12/2010

Order Total: $85.65
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

In Listing 5-9, we use the `CreateSourceQuery()` method to get access to the query used to retrieve the order item EntityCollection. Once we have the query, we apply the `Sum()` method, passing in a lambda expression that calculates the item total. The resulting sum over the collection is the order total. This entire expression is converted to the appropriate store layer commands and executed in the storage layer, saving the cost of materializing each order item.

This simple example demonstrates the flexibility of the `CreateSourceQuery()` method to modify the query used to retrieve the underlying associated entity collection. In this case, we leveraged the query without actually loading the collection.

5-8. Testing Whether an Entity Reference or Entity Collection Is Loaded

Problem

You want to test whether the related entity or entity collection is loaded in the object context.

Solution

Entity Framework exposes the IsLoaded property that, under most circumstances, is `true` if the entity or entity collection is loaded and available in the object context. IsLoaded is available on the navigation property if it is an entity collection. For entity references, IsLoaded is available on a property with the same name as the navigation property with the word "Reference" appended. For example, if the navigation property is Order, IsLoaded would be available on the OrderReference property.

To demonstrate the use of IsLoaded, let's assume you have a model like the one shown in Figure 5-9.

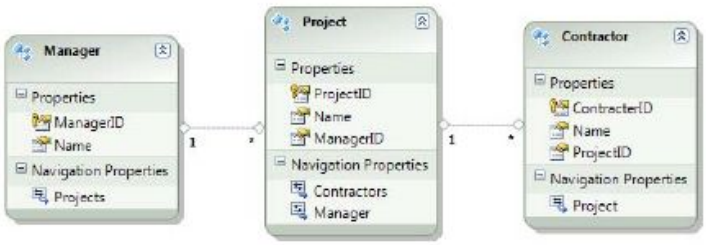


Figure 5-9. *A model for projects, managers, and contractors*

The model in Figure 5-9 represents projects, the managers for the projects, and the contractors that work on the project. To test whether an entity or entity reference is loaded into the object context, follow the pattern in Listing 5-10.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 5-10. Using IsLoaded to determine whether an entity or entity collection is in the object context

```
using (var context = new EFRecipesEntities())
{
    var man1 = new Manager { Name = "Jill Stevens" };
    var proj = new Project { Name = "City Riverfront Park", Manager = man1 };
    var con1 = new Contractor { Name = "Robert Alvert", Project = proj };
    var con2 = new Contractor { Name = "Alan Jones", Project = proj };
    var con3 = new Contractor { Name = "Nancy Roberts", Project = proj };
    context.Projects.AddObject(proj);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var project = context.Projects.Include("Manager").First();
    if (project.ManagerReference.IsLoaded)
        Console.WriteLine("Manager entity is loaded.");
    else
        Console.WriteLine("Manager entity is NOT loaded.");
    if (project.Contractors.IsLoaded)
        Console.WriteLine("Contractors are loaded.");
    else
        Console.WriteLine("Contractors are NOT loaded.");

    Console.WriteLine("Calling project.Contractors.Load()...");
    project.Contractors.Load();

    if (project.Contractors.IsLoaded)
        Console.WriteLine("Contractors are now loaded.");
    else
        Console.WriteLine("Contractors failed to load.");
}
```

The following is the output from the code in Listing 5-10:

```
Manager entity is loaded.

Contractors are NOT loaded.

Calling project.Contractors.Load()...

Contractors are now loaded.
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

We use the `Include()` method to eagerly load the project together with its related manager in the original query. After the query, we check whether the manager instance is loaded using the `project`.ManagerReference.IsLoaded property. Because this is an entity reference, the IsLoaded property is available on ManagerReference property rather than on Manager property, which is `null`.

Next, we check whether the Contractor entity collection is loaded. It is not loaded because we didn't eagerly load it with the `Include()` method nor did we load it directly (yet) with the `Load()` method. Once we use the `Load()` method, IsLoaded is set to `true`.

If lazy loading is enabled on the object context by setting DeferredLoadingEnabled to `true`, then IsLoaded is set to true when the entity or entity collection is referenced. The DeferredLoadingEnabled flag causes Entity Framework to automatically load the entity or entity collection when referenced.

When you use the `CreateSourceQuery()` method to grab the query for loading the entity or entity collection, Entity Framework will not set IsLoaded when the query is executed.

The exact meaning of IsLoaded can be a little more confusing than it seems it should be. IsLoaded is set by the results of a query, by calling the `Load()` method, or implicitly by the span of relationship keys. When you query for an entity, there is an implicit query for the key of the related entity. If the result of this implicit query is a `null` key value, then IsLoaded is set to `true`, indicating there is no related entity in the database. This is the same value for IsLoaded we would expect if we did an explicit load on the relationship and found no related entity.

5-9. Loading Related Entities Explicitly

Problem

You want to directly load related entities.

Solution

Let's say you have a model like the one in Figure 5-10.

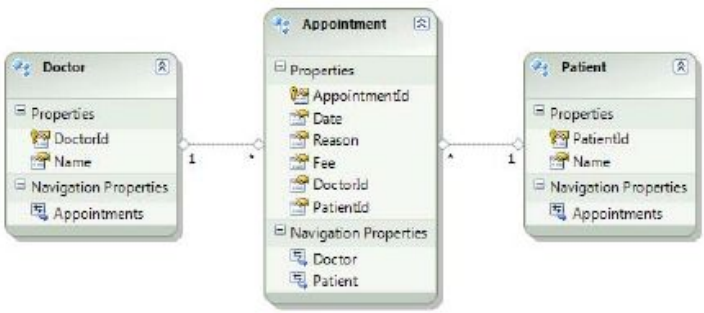


Figure 5-10. *A model for doctors, their patients, and appointments*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The model depicted in Figure 5-10 represents doctors, their patients, and appointments. To explicitly load related entities, follow the pattern in Listing 5-11.

Listing 5-11. Using the ***Load()*** *method*

```
using (var context = new EFRecipesEntities())
{
    var doc1 = new Doctor { Name = "Joan Meyers" };
    var doc2 = new Doctor { Name = "Steven Mills" };
    var pat1 = new Patient { Name = "Bill Rivers" };
    var pat2 = new Patient { Name = "Susan Stevenson" };
    var pat3 = new Patient { Name = "Roland Marcy" };
    var app1 = new Appointment { Date = DateTime.Today, Doctor = doc1,
                                 Fee = 109.92M, Patient = pat1,
                                 Reason = "Checkup" };
    var app2 = new Appointment { Date = DateTime.Today, Doctor = doc2,
                                 Fee = 129.87M, Patient = pat2,
                                 Reason = "Arm Pain" };
    var app3 = new Appointment { Date = DateTime.Today, Doctor = doc1,
                                 Fee = 99.23M, Patient = pat3,
                                 Reason = "Back Pain" };
    context.Doctors.AddObject(doc1);
    context.Doctors.AddObject(doc2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var doc = context.Doctors.First(o => o.Name == "Joan Meyers");
    if (!doc.Appointments.IsLoaded)
    {
        doc.Appointments.Load();
        Console.WriteLine("Dr. {0}'s appointments were lazy loaded.", doc.Name);
    }
    Console.WriteLine("Dr. {0} has {1} appointment(s).", doc.Name,
                      doc.Appointments.Count().ToString());

    foreach (var app in context.Appointments)
    {
        if (!app.DoctorReference.IsLoaded)
        {
            app.DoctorReference.Load();
            Console.WriteLine("Dr. {0} was lazy loaded.", app.Doctor.Name);
        }
        else
            Console.WriteLine("Dr. {0} was already loaded.", app.Doctor.Name);
    }

    Console.WriteLine("There are {0} appointments for Dr. {1}",
                      doc.Appointments.Count().ToString(), doc.Name);
    doc.Appointments.Clear();
    Console.WriteLine("Collection clear()'ed");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    Console.WriteLine("There are now {0} appointments for Dr. {1}",
                      doc.Appointments.Count().ToString(), doc.Name);
    doc.Appointments.Load();
    Console.WriteLine("Collection loaded()'ed");
    Console.WriteLine("There are now {0} appointments for Dr. {1}",
                      doc.Appointments.Count().ToString(), doc.Name);
    doc.Appointments.Load(MergeOption.OverwriteChanges);
    Console.WriteLine("Collection loaded()'ed with MergeOption.OverwriteChanges");
    Console.WriteLine("There are now {0} appointments for Dr. {1}",
                      doc.Appointments.Count().ToString(), doc.Name);
}
```

The output of the code in Listing 5-11 is the following:

```
Dr. Joan Meyers's appointments were lazy loaded.

Dr. Joan Meyers has 2 appointment(s).

Dr. Steven Mills was lazy loaded.

Dr. Joan Meyers was already loaded.

Dr. Joan Meyers was already loaded.

There are 2 appointments for Dr. Joan Meyers

Collection clear()'ed

There are now 0 appointments for Dr. Joan Meyers

Collection loaded()'ed

There are now 0 appointments for Dr. Joan Meyers

Collection loaded()'ed with MergeOPtion.OverwriteChanges

There are now 2 appointments for Dr. Joan Meyers
```

How It Works

After inserting some sample data into our database, the first bit of code retrieves an instance of the Doctor entity. It is good practice to use the IsLoaded property to check whether the entity or entity collection is already loaded. In the code, we check whether the doctor's appointments are loaded. If not, we use the **Load()** method to load them.

In the **foreach** loop, we iterate through the appointments, checking if the associated doctor is loaded. Notice in the output that one doctor was already loaded while the other one was not. This is because our first query retrieved this doctor. During the retrieval process for the appointments, Entity Framework connected the loaded instance of the doctor with her appointments. This process is informally referred to as relationship span. Relationship span will not fix up all associations. In particular, it will not tie in entities across a many-to-many association.

In the last bit of code, we print the number of appointments we have for the doctor. Then we clear the collection from the context using the **Clear()** method. The **Clear()** method empties the entity

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

collection; it does not remove the instances from memory because they are still in the object context–they are just no longer connected to this instance of the Doctor entity.

Somewhat surprisingly, after we call **Load()** to reload the appointments, we see from the output that no appointments are in our collection! What happened? It turns out that the **Load()** method is overloaded to take a parameter that controls how the loaded entities are merged into the object context. The default behavior for the **Load()** method is **MergeOption.AppendOnly**, which simply appends instances that are not already in the object context. In our case, none of the appointments was actually removed from the object context. Our use of the **Clear()** method simply removed them from the entity collection, not the object context. When we called **Load()** with the default **MergeOption.AppendOnly**, no new instances were found, so nothing was added to the entity collection. Other merge options include **NoTracking**, **OverwriteChanges**, and **PreserveChanges**. When we use the **OverwriteChanges** option, the appointments appear in the Doctor's Appointments.

The **NoTracking** option turns off object state tracking for the loaded instances. With **NoTracking**, Entity Framework will not track changes to the object and will not be aware that the object is loaded into the context. The **NoTracking** option can be used on a navigation property of an object only if the object was loaded with the **NoTracking** option. **NoTracking** has one additional side effect. If we had loaded an instance of the Doctor entity with **NoTracking**, loading the appointments with the **Load()** method would also occur with **NoTracking**, regardless of the default **AppendOnly** option.

The **OverwriteChanges** option will replace the current instance with the one found in the database. This option is particularly useful if you need to discard changes made in the context and refresh them from the database. This would be helpful, for example, in implementing an "undo" operation in an application.

The **PreserveChanges** option is, essentially, the opposite of the **OverwriteChanges** option and is typically used to force changes to objects in certain error recovery scenarios.

There are some restrictions on when you can use **Load()**. **Load()** cannot be called on an entity that is in the Added, Deleted, or Detached state.

The **Load()** method can be helpful in improving performance by restricting how much of a collection is loaded at any one time. For example, suppose our doctors had lots of appointments, but in many cases we needed to work with just a few of them. In the rare case we need the entire collection, we can simply call **Load()** to append the remaining appointment instances to the object context. This is demonstrated in the code snippet in Listing 5-12.

Listing 5-12. Code snippet demonstrating partial loading of an entity collection

```
using (var context = new EFRecipesEntities())
{
    // load the first doctor and attach just the first appointment
    var doc = context.Doctors.First(o => o.Name == "Joan Meyers");
    doc.Appointments.Attach(doc.Appointments.CreateSourceQuery().Take(1));
    Console.WriteLine("Dr. {0} has {1} appointments loaded.", doc.Name,
                      doc.Appointments.Count().ToString());

    // when we need all of the remaining appointments, simply Load() them
    doc.Appointments.Load();
    Console.WriteLine("Dr. {0} has {1} appointments loaded.", doc.Name,
                      doc.Appointments.Count().ToString());
}
```

The output of the code snippet in Listing 5-12 is the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Dr. Joan Meyers has 1 appointments loaded.

Dr. Joan Meyers has 2 appointments loaded.
```

5-10. Filtering an Eagerly Loaded Entity Collection

Problem

You want to filter an eagerly loaded collection.

Solution

Entity Framework does not support a filtering predicate with the **Include()** method, but we can accomplish the same thing by creating an anonymous type that includes the entity along with the filtered collection of related entities.

Let's assume you have a model like the one in Figure 5-11.

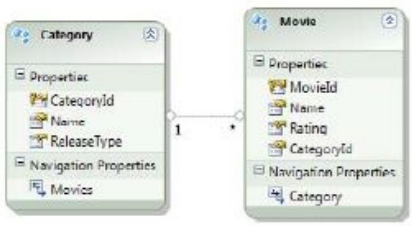


Figure 5-11. *A model for movies and their categories*

To eagerly load and filter both the categories and their associated movies, follow the pattern in Listing 5-13.

Listing 5-13. Filtering an eagerly loaded entity collection

```
using (var context = new EFRecipesEntities())
{
    var cat1 = new Category { Name = "Science Fiction", ReleaseType = "DVD" };
    var cat2 = new Category { Name = "Thriller", ReleaseType = "Blu-Ray" };
    new Movie { Name = "Return to the Moon", Category = cat1, Rating = "PG-13" };
    new Movie { Name = "Street Smarts", Category = cat2, Rating = "PG-13" };
    new Movie { Name = "Alien Revenge", Category = cat1, Rating = "R" };
    new Movie { Name = "Saturday Nights", Category = cat1, Rating = "PG-13" };
    context.Categories.AddObject(cat1);
    context.Categories.AddObject(cat2);
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    // filter on ReleaseType and Rating
    // create collection of anonymous types
    var cats = from c in context.Categories
               where c.ReleaseType == "DVD"
               select new
               {
                   category = c,
                   movies = c.Movies.Where(m => m.Rating == "PG-13")
               };

    Console.WriteLine("PG-13 Movies Released on DVD");
    Console.WriteLine("==========================");
    foreach (var c in cats)
    {
        Category category = c.category;
        Console.WriteLine("Category: {0}", category.Name);
        foreach (var m in category.Movies)
        {
            Console.WriteLine("\tMovie: {0}", m.Name);
        }
    }
}
```

The code in Listing 5-13 produces the following output:

```
PG-13 Movies Released on DVD

==========================

Category: Science Fiction

        Movie: Return to the Moon

        Movie: Saturday Nights
```

How It Works

We start off in Listing 5-13 creating and initializing the categories and movies. To keep things short, we've created only a couple of categories and four movies. Notice that we don't really need to keep references to the movies we create because we connect them immediately to their category. All we need to do is add the categories to the object context and call **SaveChanges()**. Entity Framework does the work of saving the entire object graph to the database.

In the query, we create a collection of anonymous types with the category instance and the filtered collection of movies in the category. The query also filters the category collection retrieving only categories whose movies are released on DVD. In this example, just one category was released on DVD.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Here we rely on relationship span to attach the movies to the categories. There is no need to explicitly `Attach()` the movies.

5-11. Using Relationship Span

Problem

You have a self-referencing association and you want to load all the entities and create the hierarchy without explicitly traversing the entire graph.

Solution

Suppose you have a model like the one in Figure 5-12.

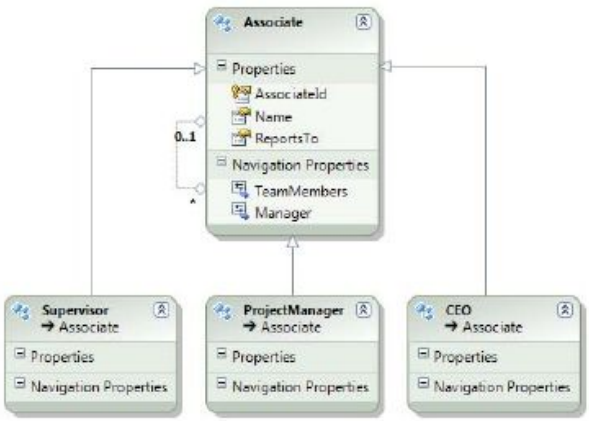


Figure 5-12. *A model with a self-referencing association*

The model in Figure 5-12 describes an associate reporting hierarchy for three types of associates: Project Manager, Supervisor, and CEO. The key feature of the model is the self-referencing association that defines the reporting hierarchy.

We have discussed in other recipes how to traverse a hierarchy such as this using recursion on both the client side and the server (database) side. Our goal here is to load the entire hierarchy letting relationship span fix up the associations to form the hierarchy.

In Listing 5-14, we use the `ToList()` method to cause the materialization of all the associates and the fix-up of the relationships. Once all the associates are in memory, we use the recursive `PrintDetails()` method to print the reporting hierarchy.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

*Listing 5-14. Using **ToList()** to cause the creation of the entire hierarchy via relationship span*

```
static void RunExample()
{
    using (var context = new EFRecipesEntities())
    {
        var ceo = new CEO { Name = "Joan Miller" };
        var super = new Supervisor { Name = "Bill Mayer", Manager = ceo };
        var pm = new ProjectManager { Name = "Jill Williams", Manager = super };
        context.Associates.AddObject(ceo);
        context.SaveChanges();
    }

    using (var context = new EFRecipesEntities())
    {
        var ceo = context.Associates.First(a => a.ReportsTo == null);
        var associates = context.Associates.ToList();
        PrintDetails(ceo);
    }
}

static void PrintDetails(Associate associate)
{
    Console.WriteLine("{0} is a {1}", associate.Name, associate.GetType().Name);
    Console.WriteLine("\t{0} reports to {1}",associate.Name,
                associate.Manager != null ? associate.Manager.Name : "No One!");
    foreach (var e in associate.TeamMembers)
    {
        PrintDetails(e);
    }
}
```

The following is the output of the code in Listing 5-14:

```
Joan Miller is a CEO

        Joan Miller reports to No One!

Bill Mayer is a Supervisor

        Bill Mayer reports to Joan Miller

Jill Williams is a ProjectManager

        Jill Williams reports to Bill Mayer
```


Licensed by Matt Allen
Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

The key to Listing 5-14 is using the **`ToList()`** method to cause the execution of the query retrieving all the associates. When the objects are materialized, not only are they brought into the object context but their associations are also fully realized. This means that the entire hierarchy is created without our code recursively loading or attaching each level of the hierarchy.

Let's look a little deeper into relationship span and how Entity Framework wires together all the associations in scenarios such as the one in Listing 5-14.

When Entity Framework loads an entity, it also loads all the associations that or 0..1 or one-to-one. Remember, associations are first-class objects just like entities. Entity Framework creates three entries in the object state manager. First, it creates the entry for the entity. Next, it creates an entry for the association. Finally, it creates an entry stub for the other end of the association that is not yet loaded.

In our example, when an Associate entity is loaded, Entity Framework creates an entry for the entity, an entry for the association, and finally, an entry stub for the Associate entity. The stub is a placeholder for the entity on the other end of the Manager relationship. The stub is a placeholder because the related associate has not yet been loaded into the object context. The stub has a valid entity key, even though the entity has not yet been loaded. It is this stub entry with the entity key that allows Entity Framework to complete the association once the related entity is loaded into the object context.

The process in which the stub entry for the association is replaced with the actual entity when the related entity is loaded is called relationship span. The result of relationship span is that the association is completed without your code explicitly connecting the entities. By retrieving all the entities as we did in Listing 5-14, Entity Framework progressively completed the associations as the entities were loaded.

5-12. Modifying Foreign Key Associations

Problem

You want to modify a foreign key association.

Solution

Entity Framework provides a couple of ways to modify a foreign key association. You can add the associated entity to a navigation property collection or assign it to a navigation property. You can also set the foreign key value with the associated entity's key value.

Suppose you have a model like the one shown in Figure 5-13.

Figure 5-13. *A model for clients and invoices*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

To modify the foreign key association between client entities and invoice entities in two different ways, do the following:

1. Right-click your project and select Add New ➤ ADO.NET Entity Data Model. Import the Client and Invoice tables. Be sure to check the Include foreign key columns in the model check box as shown in Figure 5-14. This will cause the relationships in the database that are not many-to-many to be imported as foreign key associations.
2. Use the code in Listing 5-15 to demonstrate the ways in which a foreign key association can be modified.

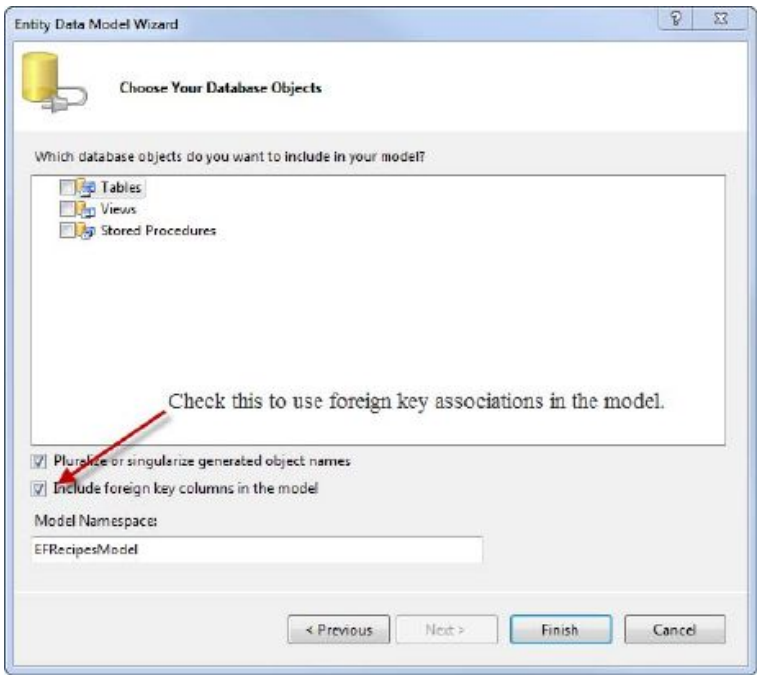


Figure 5-14. *Checking the Include foreign key columns in the model check box means that foreign key associations will be created for the imported database relationships that are not many-to-many.*

Listing 5-15. Demonstrating the ways in which a foreign key association can be modified

```
using (var context = new EFRecipesEntities())
{
    var client1 = new Client { Name = "Karen Standfield", ClientId = 1 };
    var invoice1 = new Invoice { InvoiceDate = DateTime.Parse("4/1/10"), Amount = 29.95M };
    var invoice2 = new Invoice { InvoiceDate = DateTime.Parse("4/2/10"), Amount = 49.95M };
    var invoice3 = new Invoice { InvoiceDate = DateTime.Parse("4/3/10"), Amount = 102.95M };
    var invoice4 = new Invoice { InvoiceDate = DateTime.Parse("4/4/10"), Amount = 45.99M };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    // add the invoice
    // to the client's collection
    client1.Invoices.Add(invoice1);

    // assign the foreign key
    // directly
    invoice2.ClientId = 1;

    // Attach() and existing
    // row using a "fake" entity
    context.ExecuteStoreCommand(
          "insert into chapter5.client values (2, 'Phil Marlowe')");
    var client2 = new Client { ClientId = 2 };
    context.Clients.Attach(client2);
    invoice3.Client = client2;

    // using the ClientReference
    invoice4.ClientReference.Value = client1;

    // save the changes
    context.Clients.AddObject(client1);
    context.Invoices.AddObject(invoice2);
    context.Invoices.AddObject(invoice3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var client in context.Clients)
    {
        Console.WriteLine("Client: {0}", client.Name);
        foreach (var invoice in client.Invoices)
        {
            Console.WriteLine("\t{0} for {1}",
                            invoice.InvoiceDate.ToShortDateString(),
                            invoice.Amount.ToString("C"));
        }
    }
}
```

The following is the output of the code in Listing 5-15:

```
Client: Karen Standfield

        4/1/2010 for $29.95

        4/4/2010 for $45.99

        4/2/2010 for $49.95
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Client: Phil Marlowe

        4/3/2010 for $102.95
```

How It Works

Entity Framework supports independent associations and foreign key associations. For an independent association, the association between the entities is tracked separately from the entities and the only way to change the association is through object references.

With foreign key associations, you can change the association by changing object references or by directly changing the foreign key property value. Foreign key associations are not used for many-to-many relationships.

Table 5-1 illustrates the main differences between foreign key associations and independent associations.

Table 5-1. *The Differences between Foreign Key Associations and Independent Associations*

Foreign Key Association	Independent Association
Can be set using foreign key and navigation properties	Can only be set using a navigation property
Is mapped as a property and does not require a separate mapping	Is tracked independently from the entity which means changing the association does not change the state of the entity
Data binding scenarios are easier because can bind to a property value	Data binding is complicated because you have to manually create a property that reads the foreign key value from the entity key or traverse the navigation property to load the related key
Finding the old value for a foreign key is easier because it is a property of an entity	Accessing an old relationship is complicated because relationships are tracked separately
To delete an entity that uses a foreign key association you only need the entity key	To delete an entity that uses an independent association, you need the entity key and the original values for all reference keys

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Table 5-1. Continued

Foreign Key Association	Independent Association
N-Tier scenarios are easier because you don't have to send the related end's entity key along with the entity	The client must send the related end's entity key value along with the entity When the entity is attached, Entity. Framework will create a stub entry and the update statement includes the related end's entity key
Three representations of the same association are kept in sync: the foreign key, the reference, and the collection navigation property on the other side. This is handled by Entity Framework with the default code generation, but with POCO, you need to keep these synchronized	Two representations are kept in sync: the reference and the navigation property
When you load a related entity, Entity Framework uses the foreign key value currently assigned on the entity not the foreign key value in the database	When you load a related entity, the foreign key value is read from the database and based on this value, the related entity is loaded

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 6

Beyond the Basics with Modeling and Inheritance

By now you have a solid understanding of basic modeling techniques in Entity Framework. In this chapter you will find recipes that will help you address many common and often complex modeling problems. The recipes in this chapter specifically address problems you are likely to face in modeling existing, real-world databases.

We start this chapter working with many-to-many relationships. This type of relationship is very common in many modeling scenarios in both existing databases and new projects. Next, we'll look at self-referencing relationships and explore various strategies for retrieving nested object graphs. We round out this chapter with several recipes involving more advanced modeling of inheritance and entity conditions.

6-1. Retrieving the Link Table in a Many-to-Many Association

Problem

You want to retrieve the keys in the link table that connect two entities in a many-to-many association.

Solution

Let's say you have a model with a many-to-many association between Event and Organizer entities, as shown in Figure 6-1.

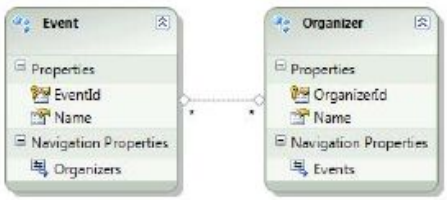


Figure 6-1. *Many-to-many association between Event and Organizer entities*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

As we illustrated in several recipes in Chapter 2, a many-to-many relationship is represented in a database using an intermediate table called a link table. The link table holds the foreign keys on each side of the relationship (see Figure 6-2). When a link table with no additional columns and the related tables are imported into Entity Framework, the Entity Data Model Wizard creates a many-to-many association between the related tables. The link table is not represented as an entity; however, it is used internally for the many-to-many association.

Figure 6-2. *A database diagram showing the EventOrganizer link table holding the foreign keys to the related Event and Organizer tables*

To retrieve the entity keys EventId and OrganizerId, we can use either a nested **from** clause or the **SelectMany()** method. Listing 6-1 shows both approaches.

*Listing 6-1. Retrieving a link table using both a nested from clause and the **SelectMany()** method*

```
using (var context = new EFRecipesEntities())
{
    var org = new Organizer { Name = "Community Charity" };
    var evt = new Event { Name = "Fundraiser" };
    org.Events.Add(evt);
    context.Organizers.AddObject(org);
    org = new Organizer { Name = "Boy Scouts" };
    evt = new Event { Name = "Eagle Scout Dinner" };
    org.Events.Add(evt);
    context.Organizers.AddObject(org);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var evsorg1 = from ev in context.Events
                  from organizer in ev.Organizers
                  select new { ev.EventId, organizer.OrganizerId };
    Console.WriteLine("Using nested from clauses...");
    foreach (var pair in evsorg1)
    {
        Console.WriteLine("EventId {0}, OrganizerId {1}",
                          pair.EventId.ToString(),
                          pair.OrganizerId.ToString());
    }

    var evsorg2 = context.Events
                         .SelectMany(e => e.Organizers,
                            (ev, org) => new { ev.EventId, org.OrganizerId });
    Console.WriteLine("\nUsing SelectManay()");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var pair in evsorg2)
    {
        Console.WriteLine("EventId {0}, OrganizerId {1}",
                          pair.EventId.ToString(), pair.OrganizerId.ToString());
    }
}
```

The output of the code in Listing 6-1 should be something like the following:

```
Using nested from clauses...

EventId 31, OrganizerId 87

EventId 32, OrganizerId 88



Using SelectManay()

EventId 31, OrganizerId 87

EventId 32, OrganizerId 88
```

How It Works

A link table is a common way of representing a many-to-many relationship between two tables in a database. Because it serves no purpose other than defining the relationship between two tables, Entity Framework represents a link table as a many-to-many association, not as a separate entity.

The many-to-many association between Event and Organizer allows easy navigation from an Event entity to the associated organizers and from an Organizer entity to all the associated events. However, you may want to retrieve just the keys in the link table. You may want to do this because the keys are themselves meaningful or you want to use these keys for operations on these or other entities. The problem here is that the link table is not represented as an entity so querying it directly is not an option. In Listing 6-1, we show a couple of ways to get just the underlying keys without materializing the entities on each side of the association.

The first approach in Listing 6-1 uses nested **from** clauses to retrieve the organizers for each event. Using the Organizers navigation property on the instances of the Event entity leverages the underlying link table to enumerate all the organizers for each of the events. We reshape the results to the pairs of corresponding keys for the entities. Finally, we iterate through the results, printing the pair of keys to the console.

In the second approach, we use the **SelectMany()** method to project the organizers for each event into the pairs of keys for the events and organizers. As with the nested **from** clauses, this approach uses the underlying link table through the Organizers navigation property. We iterate through the results in the same way as with the first approach.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

6-2. Exposing a Link Table as an Entity

Problem

You want to expose a link table as an entity instead of a many-to-many association.

Solution

Let's say your database has a many-to-many relationship between workers and tasks and looks something like the one in the database diagram in Figure 6-3.

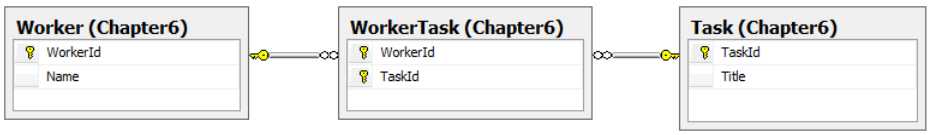


Figure 6-3. *A many-to-many relationship between workers and tasks*

The WorkerTask link table contains nothing more than the foreign keys supporting the many-to-many relationship. When we import these tables into our model, the designer will create two entities with a many-to-many association as shown in Figure 6-4.

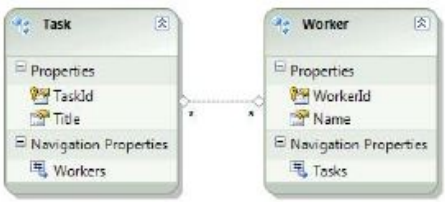


Figure 6-4. *A many-to-many association between the Worker and Task entities*

To convert the association to an entity representing the WorkerTask link table, follow these steps.

1. Delete the many-to-many association created by the designer. To delete the association, right-click the link and select Delete. When prompted to delete the WorkerTask table from the underlying store model, click No (see Figure 6-5).
2. Right-click the design surface and select Add ➤ Entity. Name the new entity **WorkerTask** and uncheck the Create key property box.
3. Right-click the WorkerTask entity and select Add ➤ Scalar property. Rename the property **WorkerId**. Repeat this step, adding TaskId scalar property.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Right-click each scalar property and select Properties. Change the type for each property from **String** to **Int32**. Mark both properties as entity key properties by right-clicking the property and selecting Entity Key.
5. Select the WorkerTask entity. In the Mapping Details window select WorkerTask in the Add Table or View drop-down control. This maps the entity to the WorkerTask table.
6. Map the WorkerId and TaskId properties to the WorkerId column and TaskId columns, respectively. See Figure 6-6.
7. Right-click the design surface and select Add ➤ Association to add a one-to-many association between the Task entity and the WorkerTask entity. Make sure that WorkerTask is on the many side of the association. Be sure to uncheck the Add foreign key properties check box because we've already created the foreign key properties. Repeat this step to create a one-to-many association between the Worker entity and the WorkerTask entity.
8. Now we need to create a referential constraint between the Task entity and the WorkerTask entity. This will complete the foreign key association between these entities. Right-click the association link and select Properties. In the properties for the association, click the Referential Constraint box. In the dialog box, choose Task as the Principal and WorkerTask as the Dependent. Choose TaskId as the Principal Key and TaskId as the Dependent Key. See Figure 6-7. Repeat this step to build the referential constraint for the association between the Worker entity and the WorkerTask entity.

The final model should look like the one in Figure 6-8.

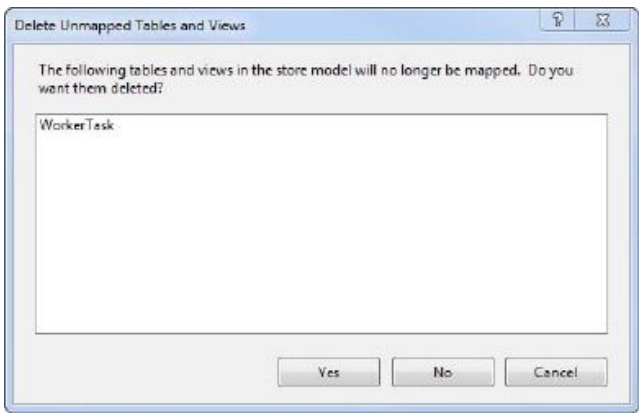


Figure 6-5. *Answer No, don't delete the underlying WorkerTask table from the store model*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

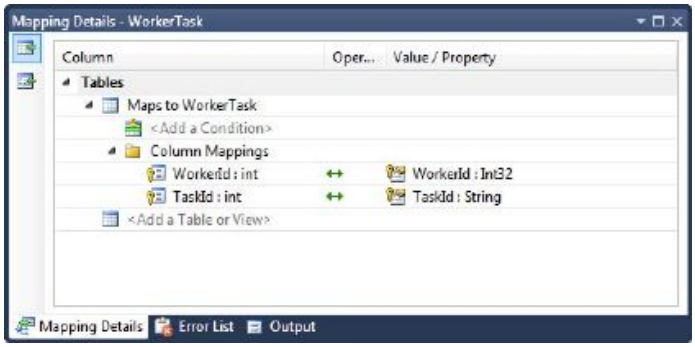


Figure 6-6. *Mapping the WorkerTask table to the WorkerTask entity in the Mapping Details window. Make sure that the WorkerId column is mapped to the WorkerId property and the TaskId property is mapped to the TaskId column.*

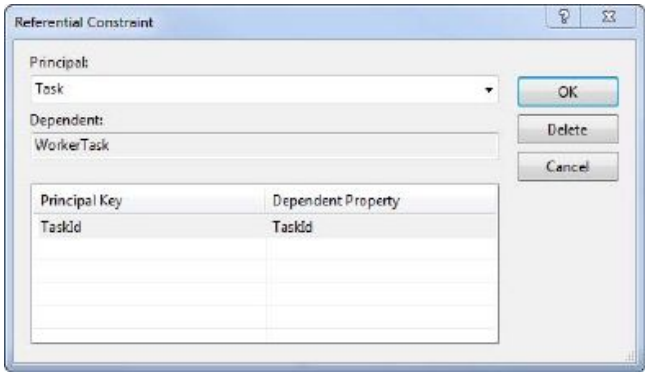


Figure 6-7. *Building the referential constraint between the Task entity and the WorkerTask entity. TaskId is the key on both sides of the constraint.*

Figure 6-8. *The completed model with the WorkerTask link table exposed as an entity in two one-to-many associations*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

When a payload-free link table in a many-to-many relationship is imported into a model, the designer will create entities to represent the related tables and will represent the link table as a many-to-many association. During the application development lifecycle, developers often find the need to add payload to the many-to-many associations that started life payload-free. In this recipe, we show how to surface the many-to-many association as a separate entity so that additional scalar properties (i.e., payload) can be added.

Of course, if your many-to-many relationship started life with a payload, the designer will create a model similar to the one shown in Figure 6-8 when it is imported. Many developers choose to assume that all many-to-many relationships will ultimately hold a payload and create a synthetic key for the link table rather than the traditional composite key formed by combining the foreign keys. This new key becomes a payload and now, when imported, the designer will start off with a model like the one shown in Figure 6-8.

The downside to our new model is that we do not have a simple way to navigate the many-to-many association. We have two one-to-many associations that require an additional hop through the linking entity. The code in Listing 6-2 demonstrates this additional bit of work on both the insert side and the query side.

Listing 6-2. Inserting into and retrieving Task and Worker entities

```
using (var context = new EFRecipesEntities())
{
    var worker = new Worker { Name = "Jim" };
    var task = new Task { Title = "Fold Envelopes" };
    var workertask = new WorkerTask { Task = task, Worker = worker };
    context.WorkerTasks.AddObject(workertask);
    task = new Task { Title = "Mail Letters" };
    workertask = new WorkerTask { Task = task, Worker = worker };
    context.WorkerTasks.AddObject(workertask);
    worker = new Worker { Name = "Sara" };
    task = new Task { Title = "Buy Envelopes" };
    workertask = new WorkerTask { Task = task, Worker = worker };
    context.WorkerTasks.AddObject(workertask);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    Console.WriteLine("Workers and Their Tasks");
    Console.WriteLine("======================");
    foreach (var worker in context.Workers)
    {
        Console.WriteLine("\n{0}'s tasks:", worker.Name);
        foreach (var wt in worker.WorkerTasks)
        {
            Console.WriteLine("\t{0}", wt.Task.Title);
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The code in Listing 6-2 produces the following output:

```
Workers and Their Tasks

=======================


Jim's tasks:

        Fold Envelopes

        Mail Letters


Sara's tasks:

        Buy Envelopes
```

6-3. Modeling a Many-to-Many, Self-Referencing Relationship

Problem

You have a table with a many-to-many relationship with itself and you want to model this table and relationship.

Solution

Let's say you have a table that has relationship to itself using a link table, as shown in Figure 6-9.

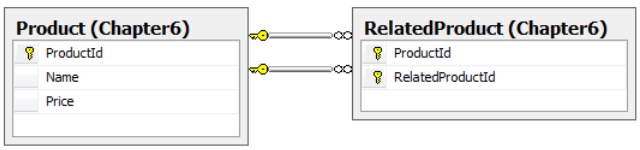


Figure 6-9. *A table with a many-to-many relationship to itself*

To create a model, do the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

1. Add a new ADO.NET Entity Data Model to your project and import the Product and RelatedProduct tables.
2. Rename the Product1 navigation property to **RelatedProducts**. Rename the Products2 navigation property to **OtherRelatedProducts**.

The completed model is shown in Figure 6-10.

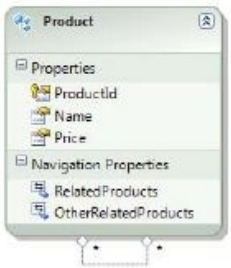


Figure 6-10. *Product entity with a many-to-many association with itself*

How It Works

As you can see, the Entity Framework designer supports a many-to-many, self-referencing association with little effort. We imported the existing table and changed the navigation property names to something more appropriate.

The code in Listing 6-3 inserts a few related products and retrieves the related products. To retrieve all the related products for a given product, we need to traverse both the RelatedProducts navigation property and the OtherRelatedProducts navigation property.

Tent is related to Ground Cover through the RelatedProducts navigation property because we added Ground Cover to Tent's RelatedProducts collection. Pole is related to Tent through Tent's OtherRelatedProducts collection because we added Tent to Pole's RelatedProducts collection. The associations go both ways. In one direction, it's a related product. In the other direction, it's an OtherRelatedProduct.

Listing 6-3. Retrieving the related products

```
using (var context = new EFRecipesEntities())
{
    var product1 = new Product { Name = "Pole", Price = 12.97M };
    var product2 = new Product { Name = "Tent", Price = 199.95M };
    var product3 = new Product { Name = "Ground Cover", Price = 29.95M };
    product2.RelatedProducts.Add(product3);
    product1.RelatedProducts.Add(product2);
    context.Products.AddObject(product1);
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    var product2 = context.Products.Include("RelatedProducts")
                                   .Include("OtherRelatedProducts")
                                   .First(p => p.Name == "Tent");
    Console.WriteLine("Product: {0} ... {1}", product2.Name,
                      product2.Price.ToString("C"));
    Console.WriteLine("Related Products");
    foreach (var prod in product2.RelatedProducts)
    {
        Console.WriteLine("\t{0} ... {1}", prod.Name, prod.Price.ToString("C"));
    }
    foreach (var prod in product2.OtherRelatedProducts)
    {
        Console.WriteLine("\t{0} ... {1}", prod.Name, prod.Price.ToString("C"));
    }
}
```

The output of Listing 6-3 is the following:

```
Product: Tent ... $199.95

Related Products

        Ground Cover ... $29.95

        Pole ... $12.97
```

The code in Listing 6-3 retrieves only the first level of related products. If we assume that the "related products" relationship is transitive, we might want to form the transitive closure. The transitive closure would be all related products regardless of how many hops away they may be. In an eCommerce application, the first level of related products could be created by product specialists. Additional levels could be derived by computing the transitive closure. The end result would allow the application to show the familiar "…you may also be interested in …" message we often see during the checkout process.

In Listing 6-4, we use a recursive method to form the transitive closure. In traversing the both the RelatedProducts and OtherRelatedProducts associations we need to be careful not to get stuck in a cycle. If product A is related to B, and B is related to A, our application would get trapped in the recursion. To detect cycles, we use a **Dictionary<>** to help prune off paths we have already traversed.

Listing 6-4. Forming the transitive closure of the "Related Products" relationship

```
static void RunExample2()
{
    using (var context = new EFRecipesEntities())
    {
        var product1 = new Product { Name = "Pole", Price = 12.97M };
        var product2 = new Product { Name = "Tent", Price = 199.95M };
        var product3 = new Product { Name = "Ground Cover", Price = 29.95M };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        product2.RelatedProducts.Add(product3);
        product1.RelatedProducts.Add(product2);
        context.Products.AddObject(product1);
        context.SaveChanges();
    }

    using (var context = new EFRecipesEntities())
    {
        var product1 = context.Products.First(p => p.Name == "Pole");
        Dictionary<int, Product> t = new Dictionary<int, Product>();
        GetRelated(product1, t);
        Console.WriteLine("Products related to {0}", product1.Name);
        foreach (var key in t.Keys)
        {
            Console.WriteLine("\t{0}", t[key].Name);
        }
    }
}

static void GetRelated(Product p, Dictionary<int, Product> t)
{
    p.RelatedProducts.Load();
    foreach (var relatedProduct in p.RelatedProducts)
    {
        if (!t.ContainsKey(relatedProduct.ProductId))
        {
            t.Add(relatedProduct.ProductId, relatedProduct);
            GetRelated(relatedProduct, t);
        }
    }
    p.OtherRelatedProducts.Load();
    foreach (var otherRelated in p.OtherRelatedProducts)
    {
        if (!t.ContainsKey(otherRelated.ProductId))
        {
            t.Add(otherRelated.ProductId, otherRelated);
            GetRelated(otherRelated, t);
        }
    }
}
```

In Listing 6-4, we use the **Load()** method (see the Recipes in Chapter 5) to ensure that the collections of related products are loaded. Unfortunately, this means we will end up with many additional round trips to the database. We might be tempted to load all the rows from the Product table up front and hope that relationship span would fix up the associations. However, relationship span will not fix up entity collections, only entity references. Because our associations are many-to-many (entity collections), we cannot rely on relationship span to help out and we have to resort to using the **Load()** method.

Following is the output of the code in Listing 6-4. From the first block of code that inserts the relationships, we can see that a Pole is related to a Tent, and a Tent is related to Ground Cover. The transitive closure for the products related to a Pole includes a Tent, Ground Cover, and Pole. Pole is included because it is on the other side of the relationship with Tent, which is a related product.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Products related to Pole

        Tent

        Ground Cover

        Pole
```

6-4. Modeling a Self-Referencing Relationship Using Table per Hierarchy Inheritance

Problem

You have a table that references itself. The table represents several different but related kinds of objects in your database. You want to model this table using table per hierarchy inheritance.

Solution

Suppose you have a table like the one in Figure 6-11 that describes some things about people. People often have a hero, perhaps the individual who inspired them the most. We can represent a person's hero with a reference to another row in the Person table.

Each person has some role in life. Some people are firefighters. Some people are teachers. Some people are retired. Of course, there could be many other roles. Information about people can be specific to their roles. A firefighter is stationed at a firehouse. A teacher teaches at a school. A retired person often has a hobby.

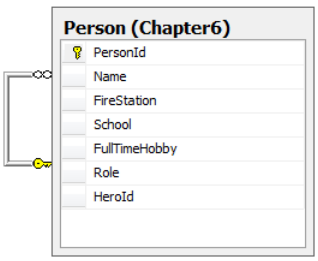


Figure 6-11. *Person table containing people with different roles*

For our example, the possible roles are firefighter (f), teacher (t), or retired (r). The role for a person is indicated by a single character in the role column.

To create a model, do the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

1. Add a new ADO.NET Entity Data Model to your project and import the Person table.
2. Right-click the design surface and select Add ➤ Entity. Name the new entity **Firefighter** and select Person as the base type. Repeat this step, creating derived entities for Teacher and Retired.
3. Move the FireStation property from the Person entity to the Firefighter entity. Move the School property from the Person entity to the Teacher entity. Finally, move the FullTimeHobby property from the Person entity to the Retired entity. You can use Cut/Paste to move these scalar properties.
4. Right-click the Person entity and view its properties. Change the Abstract value to true. This marks the Person entity as an abstract entity.
5. Rename the Person1 navigation property on the Person entity to **Fans**. This navigation property represents the person's fans (people who consider this person a hero). Rename the Person2 navigation property to **Hero**.
6. Select the Firefighter entity. In the Mapping Details window, select Add a Table or View to map the entity to the Person table.
7. In the Mapping Details window for the Firefighter entity, select Add a Condition. Add the condition for `Role = f` to conditionally map the Person table to Firefighter entity when the Role column contains the letter 'f'.
8. Repeat steps 6 and 7 for the Teacher and Retired entities using the conditions `Role = t` and `Role = r`, respectively.
9. Remove the Role property from the Person entity.

The resulting model should look like the one in Figure 6-12.

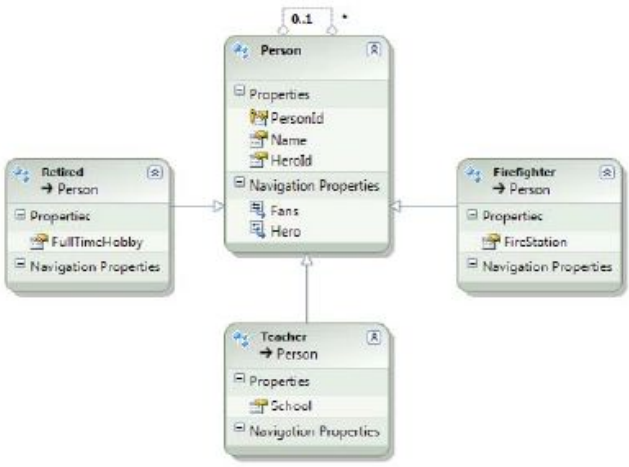


Figure 6-12. *A model for the Person type and derived types*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

The code in Listing 6-5 demonstrates inserting and retrieving from our model. We create a single instance of each of the derived types and wire in a few hero relationships. We have a teacher who is the hero of a firefighter and a retired person who is the hero of the teacher. When we set the firefighter as the hero of the retired person we introduce just enough of a cycle so that Entity Framework generates a runtime error (an UpdateException) because it cannot determine the appropriate order for inserting the rows into the table. In the code, we get around this problem by calling the **SaveChanges()** method before wiring in any of the hero relationships. Once the rows are committed to the database, and the store-generated keys are brought back into the object graph, we are free to update the graph with the relationships. Of course, these changes must be saved with a final call to **SaveChanges()**.

Listing 6-5. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var teacher = new Teacher { Name = "Susan Smith",
                                School = "Custer Baker Middle School" };
    var firefighter = new Firefighter { Name = "Joel Clark",
                                FireStation = "Midtown" };
    var retired = new Retired { Name = "Joan Collins",
                                FullTimeHobby = "Scapbooking" };
    context.People.AddObject(teacher);
    context.People.AddObject(firefighter);
    context.People.AddObject(retired);
    context.SaveChanges();
    firefighter.Hero = teacher;
    teacher.Hero = retired;
    retired.Hero = firefighter;
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    foreach(var person in context.People)
    {
        if (person.Hero != null)
            Console.WriteLine("\n{0}, Hero is: {1}", person.Name,
                                person.Hero.Name);
        else
            Console.WriteLine("{0}", person.Name);
        if (person is Firefighter)
            Console.WriteLine("Firefighter at station {0}",
                              ((Firefighter)person).FireStation);
        else if (person is Teacher)
            Console.WriteLine("Teacher at {0}", ((Teacher)person).School);
        else if (person is Retired)
            Console.WriteLine("Retired, hobby is {0}",
                              ((Retired)person).FullTimeHobby);
        Console.WriteLine("Fans:");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        foreach (var fan in person.Fans)
        {
            Console.WriteLine("\t{0}", fan.Name);
        }
    }
}
```

The output from the code in Listing 6-5 is the following:

```
Susan Smith, Hero is: Joan Collins

Teacher at Custer Baker Middle School

Fans:

        Joel Clark


Joel Clark, Hero is: Susan Smith

Firefighter at station Midtown

Fans:

        Joan Collins


Joan Collins, Hero is: Joel Clark

Retired, hobby is Scapbooking

Fans:

        Susan Smith
```

Licensed by
Matt Allen
2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

6-5. Modeling a Self-Referencing Relationship and Retrieving a Complete Hierarchy

Problem

You are using a self-referencing table to store hierarchical data. Given a record, you want to retrieve all associated records that are part of that hierarchy any level deep.

Solution

Suppose you have a Category table like the one in the database diagram in Figure 6-13.

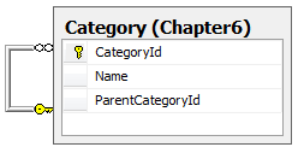


Figure 6-13. *Self-referencing Category table*

To create our model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Category table.
2. Change the navigation property Category1 to Subcategories. This property holds the collection of entities that are subcategories of this entity instance.
3. Change the Category2 navigation property to ParentCategory. This navigation property references the parent category entity instance.

The resulting model should look like the model in Figure 6-14.

Figure 6-14. *Model including the self-referencing Category entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In our model, the Category entity has a Subcategories navigation property we can use to get the collection of all the immediate subcategories of the Category. However, to access them we need to explicitly load them either using the `Load()` or the `Include()` methods. The `Load()` method requires an additional round trip to the database, while the `Include()` method provides only a predefined, limited depth.

We want to bring the entire hierarchy into the object graph as efficiently as possible. To do this, we use a Common Table Expression in a stored procedure.

To add the stored procedure to our model, do the following:

1. Create a stored procedure called `GetSubCategories()` that makes use of a Common Table Expression to return all the subcategories for a CategoryId. The stored procedure is shown in Listing 6-6.
2. Right-click the design surface and select Update Model from Database. Select the `GetSubCategories` stored procedure.
3. Now we need to add the stored procedure to the conceptual model. Open the Model Browser window. If the Model Browser window is not visible, select View ➤ Other Windows ➤ Entity Data Model Browser. Expand the Store model and the Stored Procedures levels. Right-click the `GetSubCategories` stored procedure and select Add Function Import. On the window, make sure `GetSubCategories` stored procedure is selected and leave the function Import Name as it is. Set the return type of the stored procedure to Category in the Entities box. Figure 6-15 shows the Add Function Import dialog box with the correct values.

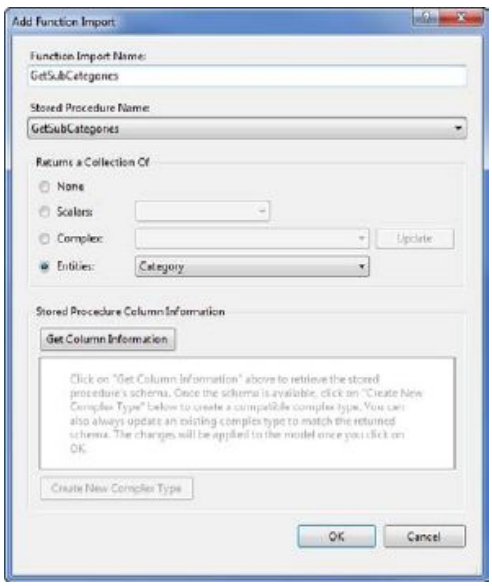


Figure 6-15. *Add Function Import dialog box, importing the* `GetSubCategories()` *stored procedure into conceptual layer*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

*Listing 6-6. The **GetSubCategories()** stored procedure that returns subcategories for a given CategoryId*

```
create proc chapter6.GetSubCategories
(@categoryid int)
as
begin
with cats as
                (
                select c1.*
                from chapter6.Category c1
                where CategoryId = @categoryid
                union all
                select c2.*
                from cats join  chapter6.Category c2 on cats.CategoryId = c2.ParentCategoryId
                )
                select * from cats where CategoryId != @categoryid
end
```

With the **GetSubCategories()** stored procedure imported into the conceptual model, Entity Framework now exposes a **GetSubCategories()** method on the object context. We can use this method to materialize our entire graph of categories and subcategories. The code in Listing 6-7 demonstrates the use of the **GetSubCategories()** method.

*Listing 6-7. Retrieving the entire hierarchy using the **GetSubCategories()** method*

```
using (var context = new EFRecipesEntities())
{
    var book = new Category { Name = "Books" };
    var fiction = new Category { Name = "Fiction", ParentCategory = book };
    var nonfiction = new Category { Name = "Non-Fiction", ParentCategory = book };
    var novel = new Category { Name = "Novel", ParentCategory = fiction };
    var history = new Category { Name = "History", ParentCategory = nonfiction };
    context.Categories.AddObject(book);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var root = context.Categories.Where(o => o.Name == "Books").First();
    Console.WriteLine("Parent category is {0}, subcategories are:", root.Name);
    foreach (var sub in context.GetSubCategories(root.CategoryId))
    {
        Console.WriteLine("\t{0}", sub.Name);
    }
}
```

The output from the code in Listing 6-7 is the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Parent category is Books, subcategories are:

        Fiction

        Non-Fiction

        History

        Novel
```

How It Works

Entity Framework supports self-referencing associations, as we have seen in Recipes 6.2 and 6.3. In these recipes, we directly loaded the entity references and collections using the **Load()** method. We cautioned, however, that each **Load()** results in a round trip to the database to retrieve an entity or entity collection. For larger object graphs, this database traffic may consume too many resources.

In this recipe, we demonstrated a slightly different approach. Rather than explicitly using **Load()** to materialize each entity or entity collection, we pushed the work off to the storage layer by using a stored procedure to recursively enumerate all the subcategories and return the collection. We used a Common Table Expression in our stored procedure to implement the recursive query. In our example, we chose to enumerate all the subcategories. You could, of course, modify the stored procedure to selectively enumerate elements of the hierarchy.

To use our stored procedure, we first imported it into the model. Then, using the Add Function Import, we added the imported stored procedure to the conceptual layer. Once added, the stored procedure was mapped by Entity Framework to a new method, **GetSubCategories()**, which was available in the data context. On the conceptual side, the stored procedure is represented in the code snippet shown in Listing 6-8.

Listing 6-8. ***GetSubCategories()*** *store procedure represented in the conceptual layer*

```
<Function Name="GetSubCategories" Aggregate="false" BuiltIn="false"
          NiladicFunction="false" IsComposable="false"
          ParameterTypeSemantics="AllowImplicitConversion" Schema="Chapter6">
  <Parameter Name="categoryid" Type="int" Mode="In" />
</Function>
```

Based on the signature of the stored procedure represented in the FunctionImport tag, Entity Framework will generate a method in the object context to make the stored procedure available to the application.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

6-6. Mapping Null Conditions in Derived Entities

Problem

You have a column in a table that allows null. You want to create a model using Table per Hierarchy inheritance with one derived type representing instances in which the column has a value and another derived type representing instances in which the column is null.

Solution

Let's say you have a table describing experimental medical drugs. The table contains a column indicating when the drug was accepted for production. Until the drug is accepted for production, it is considered experimental. Once accepted, it is considered a Medicine. We'll start with the Drug table in the database diagram in Figure 6-16.

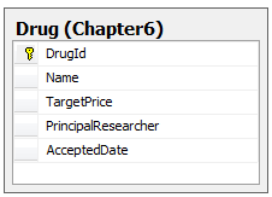


Figure 6-16. *Drug table with the nullable discriminator column, AcceptedDate*

To create a model using the Drug table, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Drug table.
2. Create the Experimental derived entity by right-clicking the design surface and selecting Add ➤ Entity. Name the entity **Experimental**. Select Drug as the base type. Repeat this step to create the Medicine entity.
3. Move the PrincipalResearcher property from the Drug entity to the Experimental entity. Move the TargetPrice and AcceptedDate properties from the Drug entity to the Medicine entity. You can use Cut/Paste to move properties between entities.
4. Mark the Drug entity as abstract. Right-click the Drug entity and select Properties. Set the Abstract property to `true`.
5. Select the Medicine entity. In the Mapping Details window, map the entity to the Drug table by selecting Add a Table or View and choosing the Drug table. Select Add a Condition and add the AcceptedDate is Not Null condition.
6. Repeat step 5 for the Experimental entity. This time, set the condition to AcceptedDate Is Null.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

7. Because all instances of Medicine will have a value for the AcceptedDate property, we need to set the Nullable attribute of this scalar property to False. This is a key step. Right-click the AcceptedDate property in the Medicine entity. Select Properties and change the Nullable attribute to False.

The completed model is shown in Figure 6-17.

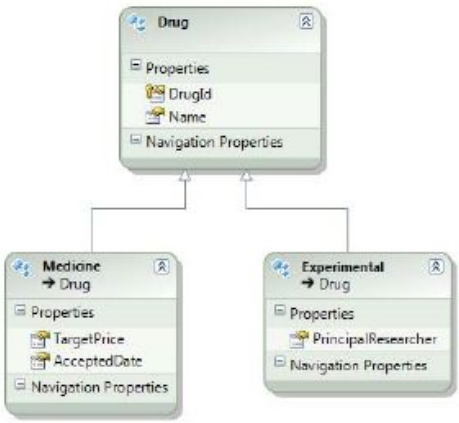


Figure 6-17. *The model for the Experimental and Medicine derived types*

How It Works

In this example, we made use of the `null` and `is not null` conditions to map a Drug without an AcceptedDate to an Experimental drug and a Drug with an AcceptedDate to a Medicine. As in many inheritance examples, we marked the base entity, Drug, as abstract because in our model we would never have an uncategorized drug.

It is interesting to note that in the Medicine entity we mapped the AcceptedDate discriminator column to a scalar property. In most scenarios, mapping the discriminator column to scalar property is prohibited. However, in this example, our use of the `null` and `is not null` conditions, as well as marking the AcceptedDate as not nullable, sufficiently constrains the values for property to allow the mapping.

In Listing 6-9, we insert a couple of Experimental drugs and query the results. We take the opportunity provided by the exposed AcceptedDate property to demonstrate one way to change an object from one derived type to another. In our case, we create a couple of Experimental drugs and then promote one of them to a Medicine.

Listing 6-9. Inserting and retrieving instances of our derived types

```
class Program
{
    …
    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        {
            var exDrug1 = new Experimental { Name = "Nanoxol",
                                 PrincipalResearcher = "Dr. Susan James" };
            var exDrug2 = new Experimental { Name = "Percosol",
                                 PrincipalResearcher = "Dr. Bill Minor" };
            context.Drugs.AddObject(exDrug1);
            context.Drugs.AddObject(exDrug2);
            context.SaveChanges();

            // Nanoxol just got approved!
            exDrug1.PromoteToMedicine(DateTime.Now, 19.99M, "Treatall");
            context.Detach(exDrug1); // better not use this instance any longer
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Experimental Drugs");
            foreach (var d in context.Drugs.OfType<Experimental>())
            {
                Console.WriteLine("\t{0} ({1})", d.Name, d.PrincipalResearcher);
            }

            Console.WriteLine("Medicines");
            foreach (var d in context.Drugs.OfType<Medicine>())
            {
                Console.WriteLine("\t{0} Retails for {1}", d.Name,
                                 d.TargetPrice.Value.ToString("C"));
            }
        }
    }
}

public partial class Experimental
{
    public void PromoteToMedicine(DateTime acceptedDate, decimal targetPrice,
                                 string marketingName)
    {
        var drug = new Medicine { DrugId = this.DrugId };
        using (var context = new EFRecipesEntities())
        {
            context.AttachTo("Drugs", drug);
            drug.AcceptedDate = acceptedDate;
            drug.TargetPrice = targetPrice;
            drug.Name = marketingName;
            context.SaveChanges();
        }
    }
}
```

We change an Experimental drug to a Medicine using the **PromoteToMedicine()** method. In the implementation of this method, we create a new Medicine instance, attach it to a new ObjectContext,

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

and initialize it with the appropriate new values. Once the new instance is attached and initialized, we use the **SaveChanges()** method on the ObjectContext to save the new instance to the database. Because the instance has the same key (DrugId) as the Experimental drug, Entity Framework generates an update statement rather than an insert statement.

We implemented the **PromoteToMedicine()** method inside the partial class **Experimental**. This allows us to seamlessly add the method to the class and provides for a much cleaner implementation.

The following is the output of the code in Listing 6-9:

```
Experimental Drugs

        Percosol (Dr. Bill Minor)

Medicines

        Treatall Retails for $19.99
```

6-7. Modeling Table per Type Inheritance Using a Non-Primary Key Column

Problem

You have one or more tables that have a one-to-one relationship to a common table using keys that are not primary keys in the tables. You want to model this using Table per Type inheritance.

Solution

Let's say your database contains the tables shown in the database diagram in Figure 6-18.

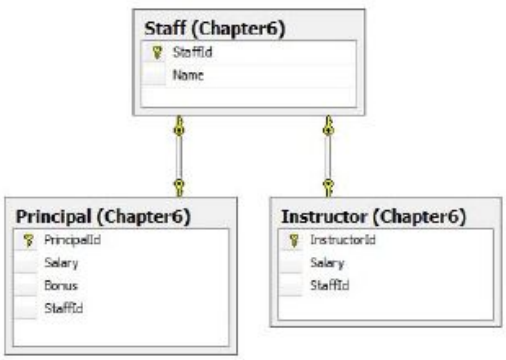


Figure 6-18. *A database diagram containing Staff, Principal, and Instructor tables*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In Figure 6-18, we have a Staff table containing the name of the staff member and two related tables containing information about Principals and Instructors. The important thing to notice here is that the Principal and Instructor tables have primary keys that are not the foreign keys for the Staff table. This type of relationship structure is not directly supported in Table per Type inheritance. For Table per Type, the related tables' primary keys must also be the foreign key for the primary (base) table. Also notice that the relationship is one-to-one. This is because we have constrained the StaffId columns in the Principal and Instructor tables to be unique by creating a unique index on this column in both tables.

To model the tables and relationships in Figure 6-18 using Table per Type inheritance, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Staff, Principal, and Instructor tables.
2. Delete the associations between the Principal and the Staff entities and between the Instructor and the Staff entities.
3. Right-click the Staff entity and choose Add ➤ Inheritance. Select Staff as the base entity and Principal as the derived entity. Repeat this step by selecting Staff as the base entity and Instructor as the derived entity.
4. Delete the StaffId property from the Instructor and Principal entities.
5. Right-click the Staff entity and choose Properties. Set the Abstract attribute to True. This marks the Staff entity as abstract.
6. Because the StaffId is not the primary key in either the Principal or the Instructor tables, we cannot use the default table mapping to map the Principal, Instructor, or Staff entities. Select each entity, view the Mapping Details window, and delete the table mapping. Repeat this for each entity.
7. Create the stored procedures in Listing 6-10. We will map these procedures to the Insert, Update, and Delete actions for the Principal and Instructor entities.
8. Right-click the design surface and select Update Model from Database. Add the stored procedures you created in step 7.
9. Select the Principal entity and view the Mapping Details window. Click the Map Entity to Functions button. This is the bottom button on the left of the Mapping Details window. Map the Insert, Update, and Delete actions to the stored procedures. Make sure you map the result columns StaffId and PrincipalId from the Insert action. See Figure 6-19.
10. Repeat step 9 for the Instructor entity. Be sure to map the result columns StaffId and InstructorId from the Insert action.
11. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor. This will close the designer and open the .edmx file in the XML editor. Scroll down to `<EntityContainerMapping>` tag in the mapping layer. Insert the QueryView in Listing 6-11 into the `<EntitySetMapping>` tag.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

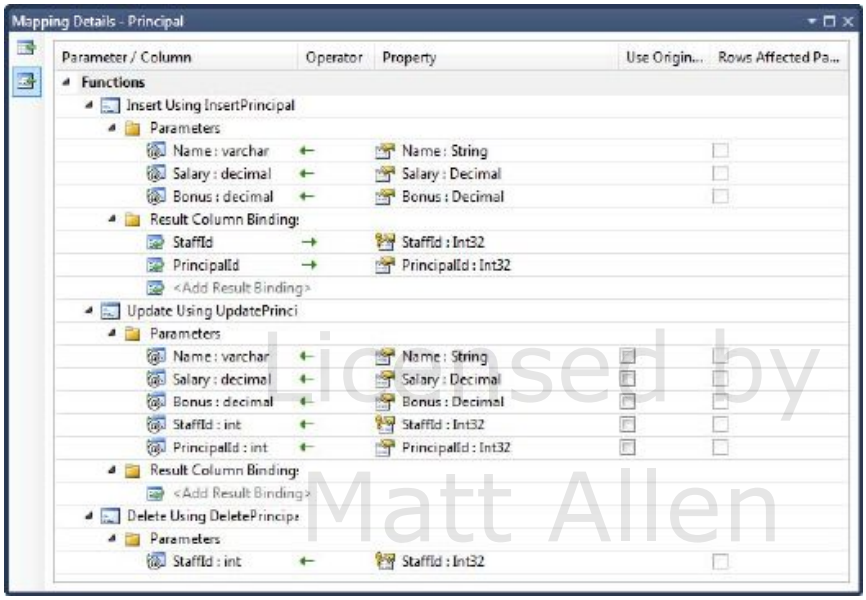


Figure 6-19. *Insert, Update, and Delete actions mapped for the Principal entity*

Licensed by Matt Allen 2175280

Listing 6-10. Stored procedures for the Insert,Update, and Delete Actions for the Instructor and Principal entities

```
create procedure [chapter6].[InsertInstructor]
(@Name varchar(50), @Salary decimal)
as
begin
                declare @staffid int
                insert into Chapter6.Staff(Name) values (@Name)
                set @staffid = SCOPE_IDENTITY()
                insert into Chapter6.Instructor(Salary,StaffId) values (@Salary,@staffid)
                select @staffid as StaffId,SCOPE_IDENTITY() as InstructorId
end
go

create procedure [chapter6].[UpdateInstructor]
(@Name varchar(50), @Salary decimal, @StaffId int, @InstructorId int)
as
begin
                update Chapter6.Staff set Name = @Name where StaffId = @StaffId
                update Chapter6.Instructor set Salary = @Salary where InstructorId =
@InstructorId
end
go
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
create procedure [chapter6].[DeleteInstructor]
(@StaffId int)
as
begin
                delete Chapter6.Staff where StaffId = @StaffId
                delete Chapter6.Instructor where StaffId = @StaffId
end
go

create procedure [Chapter6].[InsertPrincipal]
(@Name varchar(50),@Salary decimal,@Bonus decimal)
as
begin
                declare @staffid int
                insert into Chapter6.Staff(Name) values (@Name)
                set @staffid = SCOPE_IDENTITY()
                insert into Chapter6.Principal(Salary,Bonus,StaffId) values
(@Salary,@Bonus,@staffid)
                select @staffid as StaffId, SCOPE_IDENTITY() as PrincipalId
end
go

create procedure [Chapter6].[UpdatePrincipal]
(@Name varchar(50),@Salary decimal, @Bonus decimal, @StaffId int, @PrincipalId int)
as
begin
                update Chapter6.Staff set Name = @Name where StaffId = @StaffId
                update Chapter6.Principal set Salary = @Salary, Bonus = @Bonus where
PrincipalId = @PrincipalId
end
go

create procedure [Chapter6].[DeletePrincipal]
(@StaffId int)
as
begin
                delete Chapter6.Staff where StaffId = @StaffId
                delete Chapter6.Principal where StaffId = @StaffId
end
```

Listing 6-11. QueryView for the Instructor and Principal entities

```
<EntitySetMapping Name="Staffs">
    <QueryView>
        select value
        case
        when (i.StaffId is not null) then
        EFRecipesModel.Instructor(s.StaffId,s.Name,i.InstructorId,i.Salary)
        when (p.StaffId is not null) then
        EFRecipesModel.Principal(s.StaffId,s.Name,p.PrincipalId,p.Salary,p.Bonus)
        END
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        from EFRecipesModelStoreContainer.Staff as s
        left join EFRecipesModelStoreContainer.Instructor as i
        on s.StaffId = i.StaffId
        left join EFRecipesModelStoreContainer.Principal as p
        on s.StaffId = p.StaffId
    </QueryView>
</EntitySetMapping>
```

How It Works

With Table per Type inheritance, Entity Framework requires that the foreign key for the base entity's table be the primary keys in the derived entity's table. In our example, each of the tables for the derived entities have separate primary keys.

To create a Table per Type inheritance model, we started at the conceptual level by deriving the Principal and Instructor entities from the Staff entity. Next, we deleted the mappings created when we imported the table. We then used a QueryView expression to create the new mappings. Using QueryView pushed the responsibility for the Insert, Update, and Delete actions onto our code. To handle these actions, we used traditional stored procedures in the database.

We used QueryView to supply the mappings from our underlying tables to the scalar properties exposed by our derived entities. The key part of the QueryView is the case statement. There are two cases: either we have a Principal or we have an Instructor. We have an Instructor if the Instructor's StaffId is not null. Or, we have a Principal if the Principal's StaffId is not null. The remaining parts of the expression bring in the rows from the derived tables.

The code in Listing 6-12 inserts a couple of Principals and one Instructor into our database.

Listing 6-12. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var principal = new Principal { Name = "Robbie Smith",
                                    Bonus = 3500M, Salary = 48000M };
    var instructor = new Instructor { Name = "Joan Carlson",
                                    Salary = 39000M };
    context.Staffs.AddObject(principal);
    context.Staffs.AddObject(instructor);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Principals");
    Console.WriteLine("==========");
    foreach (var p in context.Staffs.OfType<Principal>())
    {
        Console.WriteLine("\t{0}, Salary: {1}, Bonus: {2}",
                          p.Name, p.Salary.ToString("C"),
                          p.Bonus.ToString("C"));
    }
    Console.WriteLine("Instructors");
    Console.WriteLine("===========");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var i in context.Staffs.OfType<Instructor>())
    {
        Console.WriteLine("\t{0}, Salary: {1}", i.Name, i.Salary.ToString("C"));
    }
}
```

The following is the output of the code in Listing 6-12:

```
Principals

==========

        Robbie Smith, Salary: $48,000.00, Bonus: $3,500.00

Instructors

===========

        Joan Carlson, Salary: $39,000.00
```

6-8. Modeling Nested Table per Hierarchy Inheritance

Problem

You want to model a table using more than one level of Table per Hierarchy Inheritance.

Solution

Suppose we have an Employee table that contains various types of employees such as Hourly and Salaried Employee, as shown in Figure 6-20.

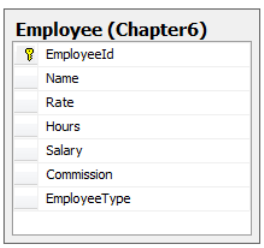


Figure 6-20. *The Employee table containing various types of employees*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The employee table contains hourly employees, salaried employees, and commissioned employees, which is a subtype of salaried employees. To model this table with derived types for the hourly and salaried employees and a commissioned employee type derived from the salaried employee, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Employee table.
2. Right-click the design surface and choose Add ➤ Entity. Name the entity **HourlyEmployee** and select Employee as the base entity. Repeat this step for the SalariedEmployee entity. Make sure you select Employee as the base entity.
3. Right-click the design surface and choose Add ➤ Entity. Name the entity **CommissionedEmployee** and select SalariedEmployee as the base type.
4. Right-click the Employee entity and choose Properties. Set the Abstract property to `true`.
5. Remove EmployeeType property from Employee. This property will serve as the discriminator column.
6. Move the Rate and Hours properties from the Employee entity to the HourlyEmployee entity. You can use Cut/Paste to move the properties. Repeat this step moving the Salary property to the SalariedEmployee entity and the Commission property to the CommissionedEmployee entity.
7. Right-click the Commission property in the CommissionedEmployee entity and select Properties. Change the nullable property to `false`.
8. Select the HourlyEmployee entity and view the Mapping Details window. Select the Employee table in Add a Table or View. Add a condition for `EmployeeType = hourly`.
9. Select the SalariedEmployee entity and view the Mapping Details window. Select the Employee table in Add a Table or View. Add a condition for `EmployeeType = salaried`. Add another condition for `Commission is null`.
10. Select the CommissionedEmployee entity and view the Mapping Details window. Select the Employee table in Add a Table or View. Add a condition for `Commission is not null`.

The completed model should look like the one in Figure 6-21.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

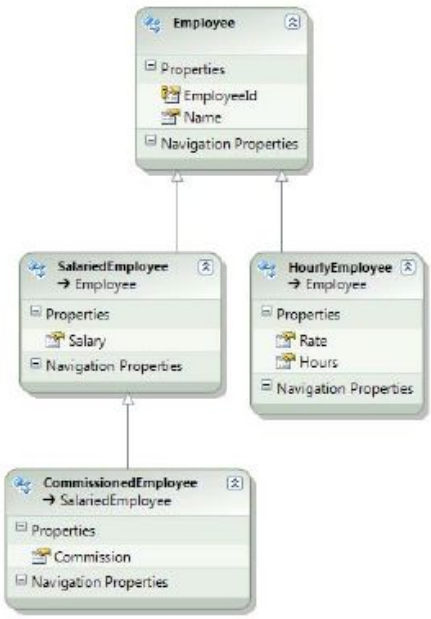


Figure 6-21. *The completed model with two levels of Table per Type inheritance*

How It Works

Table per Type inheritance is a flexible modeling technique. The depth and breadth of the inheritance tree can be reasonably large and is easily implemented. This approach is efficient because no additional tables and their required joins are involved.

We implemented the first level of the tree using simple conditions on EmployeeType. This column served as our discriminator. We ensured mutually exclusive conditions, **`is null`** and **`is not null`**, on the Commission property for the SalariedEmployee and CommissionedEmployee entities.

Listing 6-13 demonstrates inserting into and retrieving from our model.

Listing 6-13. Inserting and retrieving derived entities from Employee

```
using (var context = new EFRecipesEntities())
{
    var hourly = new HourlyEmployee { Name = "Will Smith", Hours = 39,
                                      Rate = 7.75M };
    var salaried = new SalariedEmployee { Name = "JoAnn Woodland",
                                      Salary = 65400M };
    var commissioned = new CommissionedEmployee { Name = "Joel Clark",
                                      Salary = 32500M, Commission = 20M };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.Employees.AddObject(hourly);
    context.Employees.AddObject(salaried);
    context.Employees.AddObject(commissioned);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("All Employees");
    Console.WriteLine("=============");
    foreach (var emp in context.Employees)
    {
        if (emp is HourlyEmployee)
            Console.WriteLine("{0} Hours = {1}, Rate = {2}/hour",
                              emp.Name,
                              ((HourlyEmployee)emp).Hours.Value.ToString(),
                              ((HourlyEmployee)emp).Rate.Value.ToString("C"));
        else if (emp is CommissionedEmployee)
            Console.WriteLine("{0} Salary = {1}, Commission = {2}%",
                        emp.Name,
                        ((CommissionedEmployee)emp).Salary.Value.ToString("C"),
                        ((CommissionedEmployee)emp).Commission.ToString());
        else if (emp is SalariedEmployee)
            Console.WriteLine("{0} Salary = {1}", emp.Name,
                        ((SalariedEmployee)emp).Salary.Value.ToString("C"));
    }
}
```

The output of the code in Listing 6-13 is the following:

```
All Employees

=============

Will Smith Hours = 39, Rate = $7.75/hour

JoAnn Woodland Salary = $65,400.00

Joel Clark Salary = $32,500.00, Commission = 20.00%
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

6-9. Limiting the Values Assigned to a Foreign Key

Problem

You have several foreign key columns in a table. All these foreign keys reference a primary key column in a single lookup table. You want to limit the values inserted into the foreign key columns to subsets of values that are contained in the lookup table.

Solution

Suppose you have an Order table that has foreign key columns whose values come from a single Lookup table, as shown in Figure 6-22.

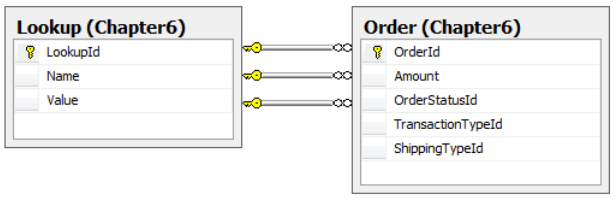
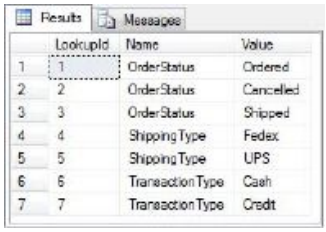


Figure 6-22. *Order table with foreign key columns referencing a Lookup table*

The columns OrderStatusId, TransactionTypeId, and ShippingTypeId are foreign keys referencing the LookupId column in the Lookup table. The database will constrain values for these columns to values that exist in the Lookup table. However, these database constraints do not limit the values to the appropriate subsets in the Lookup table.

For example, let's say the Lookup table contains the rows shown in Figure 6-23. LookupIds 1, 2, and 3 pertain to order status. LookupIds 4 and 5 are shipping types. And finally, LookupIds 6 and 7 are for transaction types. Constraints at the database layer would prevent us for inserting a row into the Order table with an OrderStatusId of 8, but it would not prevent us for inserting a row with an OrderStatusId of 7. From Figure 6-23, we know that an OrderStatusId of 7 makes no sense. The appropriate values should be either 1(Ordered), 2 (Cancelled), or 3 (Shipped).



Results | Messages

	LookupId	Name	Value
1	1	OrderStatus	Ordered
2	2	OrderStatus	Cancelled
3	3	OrderStatus	Shipped
4	4	ShippingType	Fedex
5	5	ShippingType	UPS
6	6	TransactionType	Cash
7	7	TransactionType	Credit

Figure 6-23. *A typical collection of rows for our Lookup table. Notice that there are three subsets of lookup values: one meaningful for order status, another for shipping types, and a third for transaction types.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In the database we cannot constrain the foreign key values by subset, but in Entity Framework we can build a model that does impose the limits we want. Follow these steps to create a model for the tables in Figure 6-22:

1. Add a new ADO.NET Entity Data Model to your project and import the Order and Lookup tables.
2. Delete the three associations between Order and Lookup table.
3. Right-click the design surface and choose Add ➤ Entity. Name the new entity **OrderStatus** and select Lookup as the base type.
4. Select the OrderStatus entity and view the Mapping Details window. In Add a Table or View, select the Lookup table. This maps the entity to the Lookup table. Add the condition Where `Name = OrderStatus`.
5. Repeat steps 3 and 4, creating the entities ShippingType and TransactionType. Add the conditions `Name = ShippingType`, and `Name = TransactionType`, respectively, in the Mapping Details window. In both cases, map the new entities to the Lookup table.
6. Right-click the Lookup entity and view its properties. Set the Lookup entity's Abstract property to True.
7. Right-click the Order entity and choose Add ➤ Association to create a one-to-many association between OrderStatus and Order. Set the multiplicity on the Order side to Many and the multiplicity on the OrderStatus side to One.
8. Right-click the association between Order and OrderStatus entities and view the association's properties. Click the Referential Constraint box. In the Referential Constraint dialog box, choose OrderStatus as the Principal. Set the Dependent Property to OrderStatusId.
9. Repeat steps 7 and 8 for the ShippingType and TransactionType entities.
10. Delete the Name property from the Lookup entity.

The resulting model is shown in Figure 6-24.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

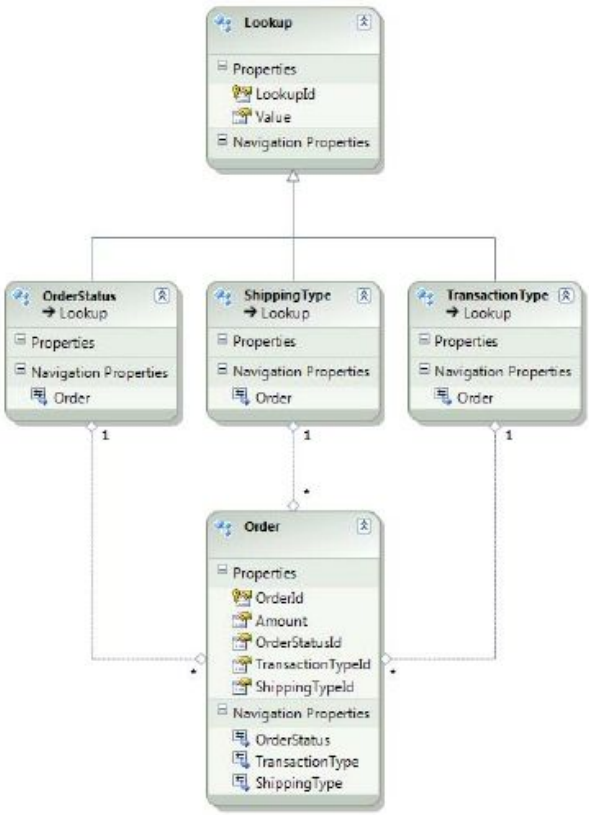


Figure 6-24. *The completed model for the Order and Lookup tables*

How It Works

It is common for a lookup table to be overloaded with semantically different subsets of values. In our example, we used just one lookup table with three different kinds of values: order status, transaction type, and shipping type. Of course, we could have created three separate lookup tables, but this approach does not scale well and often ends up cluttering an otherwise clean database design with lots of small lookup tables.

Overloading of a lookup table does introduce one problem. Most databases can constrain foreign key column values to values that are contained in the related lookup tables. However, they typically cannot constrain these values to subsets of a lookup table. In our example, the database would allow us to set the OrderStatus column of an Order to "Cash," which is a transaction type, not a valid order status.

We can address this problem at the conceptual level by introducing three entities derived from our Lookup entity. These derived entities—OrderStatus, TransactionType, and ShippingType—surface the

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

semantics of the Lookup table as strongly typed entities. This allows us to leverage the type system to enforce the finer grain constraint.

The code in Listing 6-14 demonstrates inserting and retrieving orders. Notice that before we create the orders, we grab instances of our derived Lookup entities. We use them in creating the orders.

On the query side, we use the **Include()** method to load Lookup instances together with the instance of the Order entity. This is admittedly ugly. A better approach would to be load all the possible lookup values with something as simple as **context.Lookup.ToList()**; then use the much cleaner syntax **foreach(var order in context.Order)** to iterate through the orders. This works because the **ToList()** method forces the materialization of the entire Lookup table. The entity references in each Order instance are fixed up by Entity Framework. This is commonly known as relationship span.

Listing 6-14. Inserting into and retrieving orders

```
using (var context = new EFRecipesEntities())
{
    var ordered = context.Lookups.OfType<OrderStatus>()
                                 .First(s => s.Value == "Ordered");
    var shipped = context.Lookups.OfType<OrderStatus>()
                                 .First(s => s.Value == "Shipped");
    var cash = context.Lookups.OfType<TransactionType>()
                                 .First(s => s.Value == "Cash");
    var fedex = context.Lookups.OfType<ShippingType>()
                                 .First(s => s.Value == "FedEx");
    var order = new Order { Amount = 99.97M, OrderStatus = shipped,
                            ShippingType = fedex, TransactionType = cash };
    context.Orders.AddObject(order);
    order = new Order { Amount = 29.99M, OrderStatus = ordered,
                        ShippingType = fedex, TransactionType = cash };
    context.Orders.AddObject(order);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    Console.WriteLine("Active Orders");
    Console.WriteLine("============");
    foreach (var order in context.Orders)
    {
        Console.WriteLine("\nOrder: {0}", order.OrderId.ToString());
        Console.WriteLine("Amount: {0}", order.Amount.ToString("C"));
        Console.WriteLine("Status: {0}", order.OrderStatus.Value);
        Console.WriteLine("Shipping via: {0}", order.ShippingType.Value);
        Console.WriteLine("Paid by: {0}", order.TransactionType.Value);
    }
}
```

The output of Listing 6-14 is the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Active Orders

=============

Order: 15

Amount: $99.97

Status: Shipped

Shipping via: Fedex

Paid by: Cash


Order: 16

Amount: $29.99

Status: Ordered

Shipping via: Fedex

Paid by: Cash
```

6-10. Applying Conditions in Table per Type Inheritance

Problem

You want to apply conditions while using Table per Type inheritance.

Solution

Let's say you have the two tables depicted in Figure 6-25. The Toy table describes toys a company produces. Most toys manufactured by the company are for sale. Some toys are made just to donate to worthy charities. During the manufacturing process, a toy may be damaged. Damaged toys are refurbished, and an inspector determines the resulting quality of the refurbished toy.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

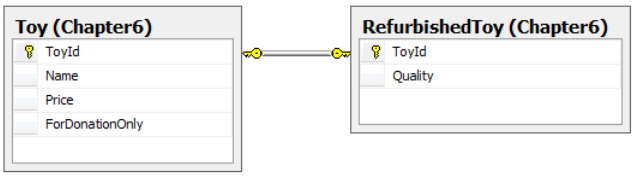


Figure 6-25. *Toy and RefurbishedToy tables with a one-to-one relationship*

The application that generates reports for the company has no need to access toys manufactured for donations. To create a model that filters out toys for donation while representing the Toy and RefurbishedToy tables using Table per Type inheritance, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Order and Lookup tables.
2. Delete the association between Toy and RefurbishedToy.
3. Right-click the Toy entity and select Add ➤ Inheritance. Select Toy as the base entity and RefurbishedToy as the derived entity.
4. Delete the ToyId property in the RefurbishedToy entity.
5. Select the RefurbishedToy entity. In the Mapping Details window, map the ToyId column to the ToyId property. This value will come from the Toy base entity.
6. Delete the ForDonationOnly scalar property from the Toy entity.
7. Select the Toy entity and view the Mapping Details window. Use Add a Table or View to map this entity to the Toy table. Add a condition When `ForDonationOnly = 0`.

The resulting model is shown in Figure 6-26.

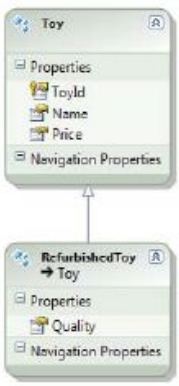


Figure 6-26. *The completed model with the Toy entity and derived RefurbishedToy entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

We limited the RefurbishedToy instances to non-donation toys by applying a condition on the base entity. This approach is useful in cases such as this in which we need to apply a permanent filter to an inheritance structure while using separate tables to implement some of the derived types.

The code in Listing 6-15 demonstrates inserting into and retrieving from our model.

Listing 6-15. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    context.ExecuteStoreCommand(@"insert into chapter6.toy
             (Name,ForDonationOnly) values ('RagDoll',1)");
    var toy = new Toy { Name = "Fuzzy Bear", Price = 9.97M };
    var refurb = new RefurbishedToy { Name = "Derby Car", Price = 19.99M,
                                      Quality = "Ok to sell" };
    context.Toys.AddObject(toy);
    context.Toys.AddObject(refurb);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("All Toys");
    Console.WriteLine("========");
    foreach (var toy in context.Toys)
    {
        Console.WriteLine("{0}", toy.Name);
    }
    Console.WriteLine("\nRefurbished Toys");
    foreach (var toy in context.Toys.OfType<RefurbishedToy>())
    {
        Console.WriteLine("{0}, Price = {1}, Quality = {2}", toy.Name,
                          toy.Price, ((RefurbishedToy)toy).Quality);
    }
}
```

The following is the output from Listing 6-15:

```
All Toys

========

Fuzzy Bear

Derby Car
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Refurbished Toys

Derby Car, Price = 19.99, Quality = Ok to sell
```

6-11. Creating a Filter on Multiple Criteria

Problem

You want to filter rows for an entity based on multiple criteria.

Solution

Let's assume we have a table that holds web orders, as shown in Figure 6-27.

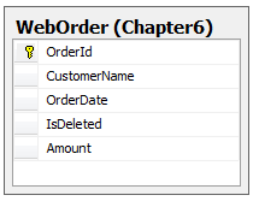


Figure 6-27. *The WebOrder table containing information about a web order*

Suppose we have a business requirement that defines instances of WebOrder as orders placed after the first day of 2007 or orders placed between 2005 and 2007 that are not deleted or orders placed before 2005 that have an order amount greater than $200. This kind of filter cannot be created using the rather limited conditions available in the Mapping Details window in the designer. One way to implement this complex filter is to use QueryView. To model this entity and implement a filter that satisfies the business requirement using QueryView, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the WebOrder table.
2. Create the stored procedures in Listing 6-16. In the next two steps, we'll map these to the insert, update, and delete actions for the WebOrder entity.
3. Right-click the design surface and select Update Model from Database. In the Update Wizard, select the InsertOrder, UpdateOrder, and DeleteOrder stored procedures.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Select the WebOrder entity and select the Map Entities to Functions button in the Mapping Details window. This button is the second of two buttons on the left side of the window. Map the InsertOrder procedure to the Insert action, the UpdateOrder procedure to the Update action, and the DeleteOrder procedure to the Delete action. The property/parameter mappings should automatically line up. However, the return value from the InsertOrder procedure must be mapped to the OrderId property. This is used by Entity Framework to get the value of the identity column OrderId after an insert. Figure 6-28 shows the correct mappings.
5. Select the table mapping (top button) in the Mapping Details window. Delete the mapping to the WebOrder table. We'll map this using QueryView.
6. Right-click the .edmx file in the Solution Explorer window and select Open With ➤ XML Editor. In the C-S mapping layer, inside the **<EntitySetMapping>** tag, enter the code shown in Listing 6-17. This is the QueryView that will map our WebOrder entity. Be careful! Changes made to the C-S mapping layer will be lost if you do another Update Model from Database.

Listing 6-16. Procedures defined in the database for the Insert, Update, and Delete actions on the WebOrder entity

```
create procedure [Chapter6].[InsertOrder]
(@CustomerName varchar(50),@OrderDate date,@IsDeleted bit,@Amount decimal)
as
begin
                insert into chapter6.WebOrder (CustomerName, OrderDate, IsDeleted, Amount)
                values (@CustomerName, @OrderDate, @IsDeleted, @Amount)
                select SCOPE_IDENTITY() as OrderId
end
go

create procedure [Chapter6].[UpdateOrder]
(@CustomerName varchar(50),@OrderDate date,@IsDeleted bit,
 @Amount decimal, @OrderId int)
as
begin
                update chapter6.WebOrder set CustomerName = @CustomerName,
        OrderDate = @OrderDate,IsDeleted = @IsDeleted,Amount = @Amount
                where OrderId = @OrderId
end
go

create procedure [Chapter6].[DeleteOrder]
(@OrderId int)
as
begin
                delete from Chapter6.WebOrder where OrderId = @OrderId
end
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

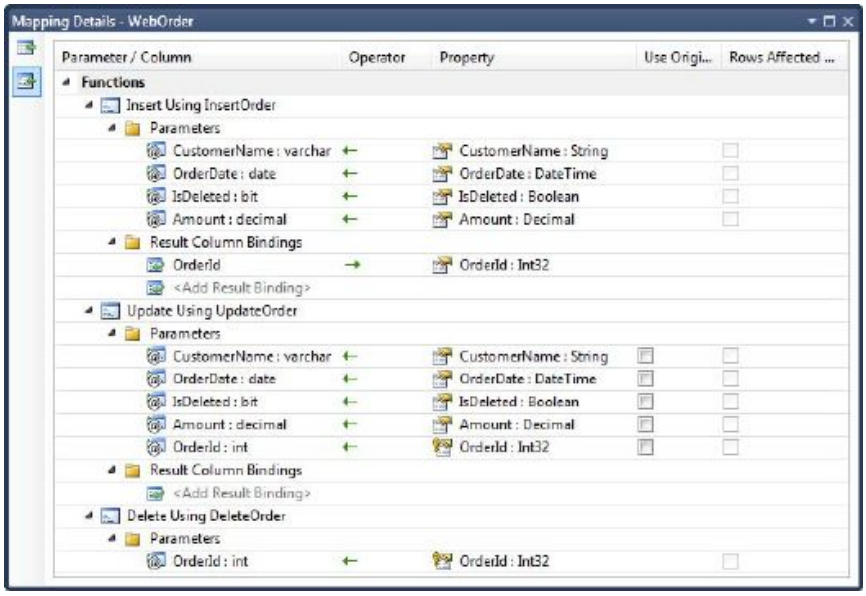


Figure 6-28. *Details for the stored procedure/action mappings*

Listing 6-17. Entity set mapping using QueryView for the WebOrder table

```
<EntitySetMapping Name="WebOrders">
  <QueryView>
    select value
    EFRecipesModel.WebOrder(o.OrderId,
    o.CustomerName,o.OrderDate,o.IsDeleted,o.Amount)
    from EFRecipesModelStoreContainer.WebOrder as o
    where (o.OrderDate > datetime'2007-01-01 00:00') ||
    (o.OrderDate between cast('2005-01-01' as Edm.DateTime) and
    cast('2007-01-01' as Edm.DateTime) and !o.IsDeleted) ||
    (o.Amount > 800 and o.OrderDate &lt;
    cast('2005-01-01' as Edm.DateTime))
  </QueryView>
</EntitySetMapping>
```

How It Works

QueryView is a read-only mapping that can be used instead of the default mapping offered by Entity Framework. When QueryView is inside of the **`<EntitySetMapping>`** tag of the mapping layer, it maps entities defined on the store model to entities defined on the conceptual model. When QueryView is inside of the **`<AssociationSetMapping>`** tag, it maps associations defined on the store model to associations defined on the conceptual model. One common use of QueryView inside of an

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

<AssociationSetMapping> tag is to implement inheritance based on conditions that are not supported by the default condition mapping.

QueryView is expressed in Entity SQL. QueryView can query only entities defined on the store model. Additionally, eSQL in QueryView does not support group by and group aggregates.

When entities are mapped using QueryView, Entity Framework is unaware of the precise implementation of the mapping. Because Entity Framework does not know the underlying columns and tables used to create instances of the entities, it cannot generate the appropriate store-level actions to insert, update, or delete the entities. Entity Framework does track changes to these entities once they are materialized, but it does not know how to modify them in the underlying data store.

The burden of implementing the insert, update, and delete actions falls onto the developer. These actions can be implemented directly in the .edmx file or they can be implemented as stored procedures in the underlying database. To tie the procedures to the actions, you need to create a **<ModificationFunctionMapping>** section. We did this in step 4 using the designer rather than directly editing the .edmx file.

If an entity mapped using QueryView has associations with other entities, those associations along with related entities also need to be mapped using QueryView. This, of course, can become rather tedious. QueryView is a powerful tool, but can rapidly become burdensome.

Some of the common use cases for using QueryView are listed as follows.

1. To define filters that are not directly supported such as greater than, less than, and so on
2. To map inheritance that is based on conditions other than is null, not null or equal to
3. To map computed columns or return subset of columns from a table or change a restriction or data type of a column like making it nullable or to surface a string column as integer
4. To map Table per Type Inheritance based on different primary and foreign key
5. To map the same column in the storage model to multiple types in the conceptual model
6. To map multiple types to the same table

Inside the QueryView in Listing 6-17, we have an Entity SQL statement that contains three parts. The first part is the **select** clause, which instantiates an instance of the WebOrder entity with a constructor. The constructor takes the property values in precisely the same order as they are defined on the conceptual model in Listing 6-18.

Listing 6-18. The definition of the WebOrder entity in the conceptual model

```
<EntityType Name="WebOrder">
  <Key>
    <PropertyRef Name="OrderId" />
  </Key>
  <Property Name="OrderId" Type="Int32" Nullable="false"
            annotation:StoreGeneratedPattern="Identity" />
  <Property Name="CustomerName" Type="String" Nullable="false"
            MaxLength="50" Unicode="false" FixedLength="false" />
  <Property Name="OrderDate" Type="DateTime" Nullable="false" />
  <Property Name="IsDeleted" Type="Boolean" Nullable="false" />
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
  <Property Name="Amount" Type="Decimal" Nullable="false"
            Precision="18" Scale="2" />
</EntityType>
```

Notice that, in the Entity SQL in Listing 6-17, we fully qualified the conceptual namespace EFRecipesModel when creating an instance of the WebOrder entity. However, in the **from** clause, we also fully qualified the store container, EFRecipesModelStoreContainer.

The final section of the Entity SQL expression includes the **where** clause that, of course, is the whole reason for using a QueryView in this example. Although the **where** clause can be arbitrarily complex, it is subject to the restrictions for Entity SQL in QueryView as noted above.

The code in Listing 6-19 demonstrates inserting and retrieving WebOrders in our model.

Listing 6-19. Inserting and retrieving WebOrder entities

```
using (var context = new EFRecipesEntities())
{
    var order = new WebOrder {CustomerName = "Jim Allen",
                              OrderDate = DateTime.Parse("5/3/2009"),
                              IsDeleted = false, Amount = 200};
    context.WebOrders.AddObject(order);
    order = new WebOrder { CustomerName = "John Stevens",
                           OrderDate = DateTime.Parse("1/1/2006"),
                           IsDeleted = false, Amount = 400 };
    context.WebOrders.AddObject(order);
    order = new WebOrder { CustomerName = "Russel Smith",
                           OrderDate = DateTime.Parse("1/3/2006"),
                           IsDeleted = true, Amount = 500 };
    context.WebOrders.AddObject(order);
    order = new WebOrder { CustomerName = "Mike Hammer",
                           OrderDate = DateTime.Parse("3/6/2006"),
                           IsDeleted = true, Amount = 1800 };
    context.WebOrders.AddObject(order);
    order = new WebOrder { CustomerName = "Steve Jones",
                           OrderDate = DateTime.Parse("1/1/2003"),
                           IsDeleted = true, Amount = 600 };
    context.WebOrders.AddObject(order);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Orders");
    Console.WriteLine("======");
    foreach (var order in context.WebOrders)
    {
        Console.WriteLine("\nCustomer: {0}", order.CustomerName);
        Console.WriteLine("OrderDate: {0}", order.OrderDate.ToShortDateString());
        Console.WriteLine("Is Deleted: {0}", order.IsDeleted.ToString());
        Console.WriteLine("Amount: {0}", order.Amount.ToString("C"));
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The output of the code in Listing 6-19 follows. Notice that only customers that meet the criteria we defined in the Entity SQL expression inside the QueryView are displayed.

```
Orders...


Customer: John Stevens

Order Date: 1/1/2006

Is Deleted: False

Amount: $400.00


Customer: Jim Allen

Order Date: 5/3/2009

Is Deleted: False

Amount: $200.00


Customer: Mike Hammer

Order Date: 6/3/2004

Is Deleted: True

Amount: $1,800.00
```

6-12. Using Complex Conditions with Table per Hierarchy Inheritance

Problem

You want to model a table using Table per Hierarchy inheritance by applying conditions more complex than those supported directly by Entity Framework.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Suppose we have a Member table, as depicted in Figure 6-29. The Member table describes members in our club. In our model, we want to represent adult members, senior members, and teen members as derived types using Table per Type inheritance.

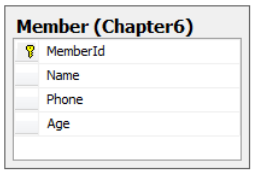


Figure 6-29. *The Member table describing members in our club*

Licensed by Matt Allen 2175280

Entity Framework supports Table per Hierarchy Inheritance based on the conditions `=`, `is null`, and `is not null`. Simple expressions such as `<`, `between`, and `>` are not supported. In our case, a member whose age is less than 20 is a teen (the minimum age in our club is 13). A member between the age of 20 and 55 is an adult. And, as you might expect, a member over the age of 55 is a senior. To create a model for the member table and the three derived types, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Member table.
2. Right-click the Member entity and select Properties. Set the Abstract attribute to `true`. This marks the Member entity as abstract.
3. Create the stored procedures in Listing 6-20. We will use them to handle the Insert, Update, and Delete actions on the entities we'll derive from the Member entity.
4. Right-click the design surface and select Update Model from Database. Select the stored procedures you created in step 3.
5. Right-click the design surface and select Add ➤ Entity. Name the new entity **Teen** and set the base type to **Member**. Repeat this step, creating the derived entities Adult and Senior.
6. Select the Member entity and view the Mapping Details window. Click Maps to Member, and select <Delete>. This deletes the mappings to the Member table.
7. Select the Teen entity and view the Mapping Details window. Click the Map Entity to Functions button. This is the bottom button on the left of the Mapping Details window. Map the stored procedures to the corresponding Insert, Update, and Delete actions. The parameter/property mappings should automatically populate. Make sure you set the Result Column Bindings to map the return value to the MemberId property for the Insert action. This identity column is generated on the database side. See Figure 6-30.
8. Repeat step 7 for the Adult and Senior entities.
9. Right-click the .edmx file in the Solution Explorer window and select Open With ➤ XML Editor. This will open the .edmx file in the XML editor.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10. In the C-S mapping section, inside the **<EntityContainerMapping>** tag, enter the QueryView code shown in Listing 6-21.

Listing 6-20. Stored procedures for the Insert, Update, and Delete actions

```
create procedure [chapter6].[InsertMember]
(@Name varchar(50), @Phone varchar(50), @Age int)
as
begin
                insert into Chapter6.Member (Name, Phone, Age)
                values (@Name,@Phone,@Age)
                select SCOPE_IDENTITY() as MemberId
end
go

create procedure [chapter6].[UpdateMember]
(@Name varchar(50), @Phone varchar(50), @Age int, @MemberId int)
as
begin
                update Chapter6.Member set Name=@Name, Phone=@Phone, Age=@Age
                where MemberId = @MemberId
end
go

create procedure [chapter6].[DeleteMember]
(@MemberId int)
as
begin
                delete from Chapter6.Member where MemberId = @MemberId
end
```

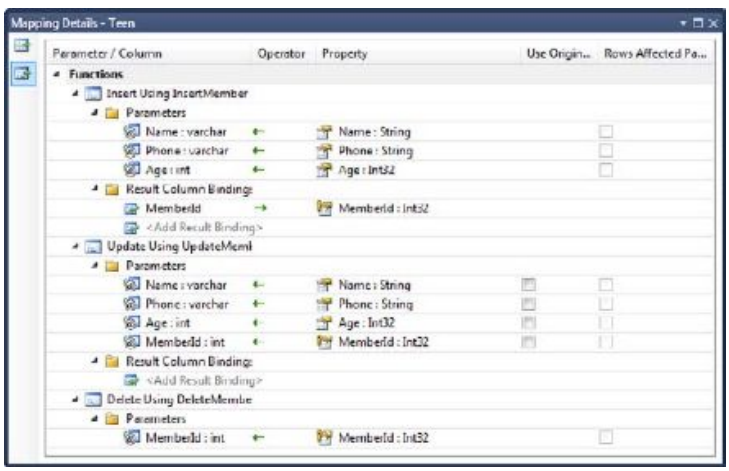


Figure 6-30. *Mapping the Insert, Update, and Delete actions for the Teen entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 6-21. QueryView for mapping the Member table to the derived types Teen, Adult, and Senior

```
<EntitySetMapping Name="Members">
  <QueryView>
    select value
    case
    when m.Age &lt; 20 then
    EFRecipesModel.Teen(m.MemberId,m.Name,m.Phone,m.Age)
    when m.Age between 20 and 55 then
    EFRecipesModel.Adult(m.MemberId,m.Name,m.Phone,m.Age)
    when m.Age > 55 then
    EFRecipesModel.Senior(m.MemberId,m.Name,m.Phone,m.Age)
    end
    from EFRecipesModelStoreContainer.Member as m
  </QueryView>
</EntitySetMapping>
```

The resulting model should look like the one in Figure 6-31.

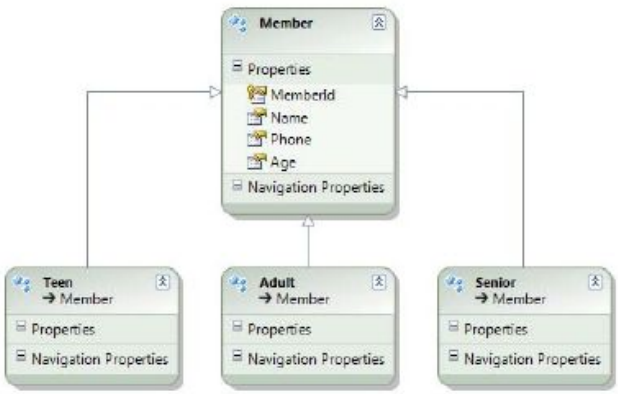


Figure 6-31. *The resulting model with Member and the three derived types: Senior, Adult, and Teen*

How It Works

Entity Framework supports only a limited set of conditions when modeling Table per Hierarchy inheritance. In this recipe, we extended the conditions using QueryView to define our own mappings between the underlying Member table and the derived types Senior, Adult, and Teen. This is shown in Listing 6-21.

Unfortunately, QueryView comes at a price. Because we have defined the mappings ourselves, we also take on the responsibility for implementing the Insert, Update, and Delete actions for the derived types. This is not too difficult in our case.

In Listing 6-20, we defined the procedures to handle the Insert, Delete, and Update actions. We need to create only one set because these actions target the underlying Member table. In this recipe, we

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

implemented them as stored procedures in the underlying database. We could have implemented in the .edmx file.

Using the designer, we mapped the procedures to the Insert, Update, and Delete actions for each of the derived types. This completes the extra work we need to do when we use QueryView.

The code in Listing 6-22 demonstrates inserting into and retrieving from our model. Here we insert one instance of each of our derived types. On the retrieval side, we print the members together with their phone number, unless the member is a Teen.

Listing 6-22. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var teen = new Teen { Name = "Steven Keller", Age = 17,
                          Phone = "817 867-5309" };
    var adult = new Adult { Name = "Margret Jones", Age = 53,
                            Phone = "913 294-6059" };
    var senior = new Senior { Name = "Roland Park", Age = 71,
                              Phone = "816 353-4458" };
    context.Members.AddObject(teen);
    context.Members.AddObject(adult);
    context.Members.AddObject(senior);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Club Members");
    Console.WriteLine("============");
    foreach(var member in context.Members)
    {
        bool printPhone = true;
        string str = string.Empty;
        if (member is Teen)
        {
            str = " a Teen";
            printPhone = false;
        }
        else if (member is Adult)
            str = "an Adult";
        else if (member is Senior)
            str = "a Senior";
        Console.WriteLine("{0} is {1} member, phone: {2}",member.Name,
                          str, printPhone ? member.Phone : "unavailable");
    }
}
```

The following is the output from the code in Listing 6-22:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Members of our club

==================

Steven Keller is a Teen member, phone: unavailable

Margret Jones is an Adult member, phone: 913 294-6059

Roland Park is a Senior member, phone: 816 353-4458
```

It is important to note here that no design time or even runtime checking is done to verify the ages for the derived types. It is entirely possible to create an instance of the Teen type and set the age property to 74—clearly not a teen. On the retrieval side, however, this row will be materialized as a Senior member; a situation likely offensive to our Teen member.

We can introduce validation before changes are committed to the data store. To do this, register for the **SavingChanges** event when the context is created. We wire this event to our code that performs the validation. This code is shown in Listing 6-23.

Listing 6-23. Handling validation in the SavingChanges event

```
public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges += new EventHandler(Validate);
    }

    public void Validate(object sender, EventArgs e)
    {
        var entities = this.ObjectStateManager
                           .GetObjectStateEntries(EntityState.Added |
                                                  EntityState.Modified)
                           .Select(et => et.Entity as Member);
        foreach (var member in entities) {
            if (member is Teen && member.Age > 19) {
                throw new ApplicationException("Entity validation failed");
            }
            else if (member is Adult && (member.Age < 20 || member.Age >= 55)) {
                throw new ApplicationException("Entity validation failed");
            }
            else if (member is Senior && member.Age < 55) {
                throw new ApplicationException("Entity validation failed");
            }
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In Listing 6-23, when `SaveChanges()` is called, our `Validate()` method checks each entity that has either been added or modified. For each of these, we verify that the age property is appropriate for the type of the entity. When we find a validation error, we simply throw an exception.

We have several recipes in Chapter 12 that focus on handling events and validating objects before they are committed to the database.

6-13. Modeling Table per Concrete Type Inheritance

Problem

You have two or more tables with similar schema and data and you want to model these tables as types derived from a common entity using Table per Concrete Type inheritance.

Solution

Let's assume we have the tables shown in Figure 6-32.

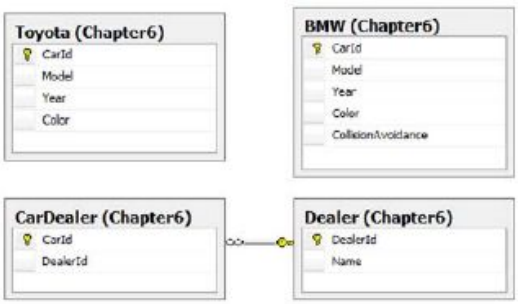


Figure 6-32. *Tables Toyota and BMW with similar structure that will become derived types of the Car entity*

In Figure 6-32, the tables Toyota and BMW that have similar schema and represent similar data. The BMW table has an additional column with a bit value indicating whether the instance has the collision avoidance feature. We want to create a model with a base entity holding the common properties of the Toyota and BMW tables. Additionally, we want to represent the one-to-many relationship between the car dealer and cars he holds in inventory. Figure 6-33 shows the final model.

To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Toyota, BMW, CarDealer, and Dealer tables.
2. Right-click the design surface and select Add ➤ Entity. Name the new entity **Car** and unselect the Create key property check box.
3. Right-click the Car entity and view its properties. Set the Abstract property to `true`.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Move the common properties of the Toyota and BMW entities to the Car entity. You can use Cut/Paste to move these properties. Make sure that only the CollisionAvoidance property remains with the BMW entity and the Toyota entity has no properties. Both of these entities will inherit these common properties from the Car entity.
5. Right-click the Car entity and select Add ➤ Inheritance. Set the base entity as Car and the derived entity as BMW.
6. Repeat step 5, but this time set the Toyota as the derived entity.
7. Right-click the CarDealer entity and select Delete. When prompted to delete the CarDealer table from the store model, select No.
8. Right-click the design surface and select Add ➤ Association. Name the association CarDealer. Select Dealer on the left with a multiplicity of one. Select Car on the right with a multiplicity of many. Name the navigation property on the Car side Dealer. Name the navigation property on the Dealer side Cars. Be sure to uncheck the Add foreign key properties.
9. Select the association and view the Mapping Details window. Select CarDealer in the Add a Table or View drop-down menu. Make sure the DealerId property maps to the DealerId column and the CarId property maps to the CarId column.
10. Right-click the .edmx file and select Open With ➤ XML Editor. Edit the mapping section with the changes shown in Listing 6-24 for the BMW and Toyota entities.

Listing 6-24. Mapping the BMW and Toyota tables

```
<EntitySetMapping Name="Cars">
  <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.BMW)">
    <MappingFragment StoreEntitySet="BMW">
      <ScalarProperty Name="CollisionAvoidance"
                      ColumnName="CollisionAvoidance" />
      <ScalarProperty Name="CarId" ColumnName="CarId"/>
      <ScalarProperty Name="Model" ColumnName="Model"/>
      <ScalarProperty Name="Year" ColumnName="Year"/>
      <ScalarProperty Name="Color" ColumnName="Color"/>
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Toyota)">
    <MappingFragment StoreEntitySet="Toyota">
      <ScalarProperty Name="CarId" ColumnName="CarId"/>
      <ScalarProperty Name="Model" ColumnName="Model"/>
      <ScalarProperty Name="Year" ColumnName="Year"/>
      <ScalarProperty Name="Color" ColumnName="Color"/>
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The resulting model is shown in Figure 6-33.

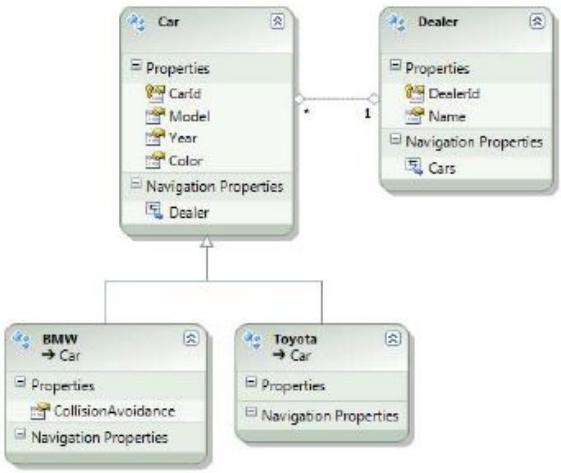


Figure 6-33. *The completed model with the derived entities BMW and Toyota represented in the database as separate tables*

How It Works

Table per Concrete Type is an interesting inheritance model in that it allows each derived entity to map to separate physical tables. From a practical perspective, the tables need to share at least some part of a common schema. This common schema is mapped in the base entity while the additional schema parts are mapped in the derived entities. For Table per Concrete Type inheritance to work properly, the entity key must be unique across the tables.

The base entity is marked abstract and is not mapped to any table. In Table per Concrete Type, only the derived entities are mapped to tables.

In our example, we marked the Car entity as abstract and did not map it to any table. In the mapping in Listing 6-24, notice that we mapped only the derived entities BMW and Toyota. We moved all the common properties (CarId, Model, Year, and Color) to the base entity. The derived entities contained only the properties unique to the entity. For instance, the BMW entity has the additional CollisionAvoidance property.

Because the entities Toyota and BMW derived from the Car entity, they became part of the same Cars entity set. This means that the CarId entity key must be unique within the entity set that now contains all the derived entities. Because the entities are mapped to different tables, it is possible that we can have collisions in the keys. To avoid this, we set the CarId column in each table as an identity column. For the BMW table, we set the initial seed to 1 with an increment of 2. This will create odd values for the CarId key. For the Toyota table, we set the initial seed to 2 with an increment of 2. This will create event values for the CarId key.

When modeling relationships in Table per Concrete Type inheritance, it is better to define them at the derived type rather than at the base type. This is because the Entity Framework runtime would not know which physical table represents the other end of the association. In our example, of course, we

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

provided a separate table (CarDealer) that contains the relationship. This allowed us to model the relationship at the base entity by mapping the association to the CarDealer table.

There are many practical applications of Table per Concrete Type inheritance. Perhaps the most common is in working with archival data. Imagine you have a several years worth of orders for your eCommerce site. At the end of each year, you archive the orders for the previous 12 months in an archive table and start the New Year with an empty table. With Table per Concrete Type inheritance, you can model the current and archived orders using the approach demonstrated here.

Table per Concrete Type inheritance has a particularly important performance advantage over other inheritance models. When querying a derived type, the generated query targets the specific underlying table without the additional joins of Table per Type inheritance or the filtering of Table per Hierarchy. For large datasets or models with several derived types, this performance advantage can be significant.

The disadvantages of Table per Concrete Type inheritance include the overhead of potentially duplicate data across tables and the complexity of insuring unique keys across the tables. In an archival scenario, data is not duplicated but simply spread across multiple tables. In other scenarios, data (properties) may be duplicated across the tables.

The code in Listing 6-25 demonstrates inserting into and retrieving from our model.

Listing 6-25. Inserting into and querying our model

```
using (var context = new EFRecipesEntities())
{
    var d1 = new Dealer { Name = "All Cities Toyota" };
    var d2 = new Dealer { Name = "Southtown Toyota" };
    var d3 = new Dealer { Name = "Luxury Auto World" };
    var c1 = new Toyota { Model = "Camry", Color = "Green",
                          Year = "2010", Dealer = d1 };
    var c2 = new BMW { Model = "310i", Color = "Blue",
                       CollisionAvoidance = true,
                       Year = "2010", Dealer = d3 };
    var c3 = new Toyota { Model = "Tundra", Color = "Blue",
                       Year = "2010", Dealer = d2 };
    context.Dealers.AddObject(d1);
    context.Dealers.AddObject(d2);
    context.Dealers.AddObject(d3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    Console.WriteLine("Dealers and Their Cars");
    Console.WriteLine("=====================");
    foreach (var dealer in context.Dealers)
    {
        Console.WriteLine("\nDealer: {0}", dealer.Name);
        foreach(var car in dealer.Cars)
        {
            string make = string.Empty;
            if (car is Toyota)
                make = "Toyota";
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            else if (car is BMW)
                make = "BMW";
            Console.WriteLine("\t{0} {1} {2} {3}", car.Year,
                              car.Color, make, car.Model);
        }
    }
}
```

The output of the code in Listing 6-25 is the following:

```
Dealers and Their Cars

======================



Dealer: Luxury Auto World

        2010 Blue BMW 310i



Dealer: Southtown Toyota

        2010 Blue Toyota Tundra



Dealer: All Cities Toyota

        2010 Green Toyota Camry
```

6-14. Applying Conditions on a Base Entity

Problem

You want to derive a new entity from a base entity that currently exists in a model and continue to allow the base entity to be instantiated.

Solution

Let's assume you have a model like the one shown in Figure 6-34.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

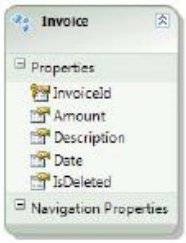


Figure 6-34. *Our model with the Invoice entity*

This model contains a single Invoice entity. We want to derive a new entity that represents deleted invoices. This will allow us to more cleanly separate business logic that operates on active invoices differently than deleted invoices. To add the derived entity, do the following:

1. View the Mapping Details window for the Invoice entity. Add a condition on the IsDeleted column to map the entity when the column is 0 as shown in Figure 6-35.
2. Now that the IsDeleted column is used in a condition, we need to remove it from the scalar properties for the entity. Right-click the IsDeleted property in the entity and select Delete.
3. Right-click the design surface and select Add ➤ Entity. Name the new entity DeletedInvoice and select Invoice as the base type.
4. View the Mapping Details window for the DeletedInvoice entity. Map the entity to the Invoice table. Add a condition on the IsDeleted column to map the entity when the column is 1 as shown in Figure 6-36.

The final model with the Invoice entity and the derived DeletedInvoice entity is shown in Figure 6-37.

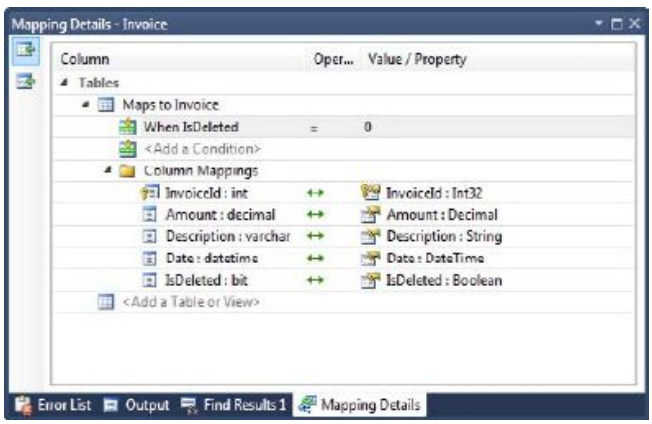


Figure 6-35. *Mapping the Invoice entity when the IsDeleted column is 0*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

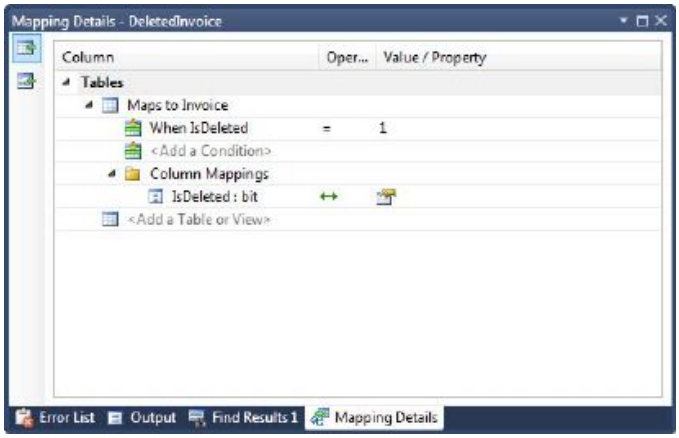


Figure 6-36. *Mapping the DeletedInvoice entity to the Invoice table when the IsDeleted column is 1*

Figure 6-37. *Our completed model with the Invoice entity and the DeletedInvoice entity*

How It Works

There are two different ways to model our invoices and deleted invoices. The approach we've shown here is only recommended if you have an existing model and code base and would like to add the DeletedInvoice derived type with as little impact as possible to the existing code. For a new model, it would be better to derive an ActiveInvoice type and a DeletedInvoice type from the Invoice base type. In this approach, you would mark the base type as abstract.

Using the approach we've shown here, you could can determine, as we do in the code in Listing 6-26, if the entity is a DeletedInvoice either by casting or by using the `OfType<>()` method. However, you can't select for the Invoice entity alone. This is the critical drawback to the approach we've shown here.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The approach you should use for new code is to derive two new entities: ActiveInvoice and DeleteInvoice. With these two sibling types, you can use either casting or the **OfType<>()** method to operate on either type uniformly.

*Listing 6-26. Using the **as** operator to determine if we have an Invoice or DeletedInvoice*

```
using (var context = new EFRecipesEntities())
{
    context.Invoices.AddObject(new Invoice { Amount = 19.95M,
                                 Description = "Oil Change",
                                 Date = DateTime.Parse("4/11/10") });
    context.Invoices.AddObject(new Invoice { Amount = 129.95M,
                                 Description = "Wheel Alignment",
                                 Date = DateTime.Parse("4/01/10") });
    context.Invoices.AddObject(new DeletedInvoice { Amount = 39.95M,
                                 Description = "Engine Diagnosis",
                                 Date = DateTime.Parse("4/01/10") });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var invoice in context.Invoices)
    {
        var isDeleted = invoice as DeletedInvoice;
        Console.WriteLine("{0} Invoice",
                          isDeleted == null ? "Active" : "Deleted");
        Console.WriteLine("Description: {0}", invoice.Description);
        Console.WriteLine("Amount: {0}", invoice.Amount.ToString("C"));
        Console.WriteLine("Date: {0}", invoice.Date.ToShortDateString());
        Console.WriteLine();
    }
}
```

The following is the output of the code in Listing 6-26:

```
Active Invoice

Description: Oil Change

Amount: $19.95

Date: 4/11/2010



Active Invoice

Description: Wheel Alignment
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Amount: $129.95

Date: 4/1/2010


Deleted Invoice

Description: Engine Diagnosis

Amount: $39.95

Date: 4/1/2010
```

6-15. Creating Independent and Foreign Key Associations

Problem

You want to use Model First to create both independent and foreign key associations.

Solution

1. Add a new ADO.NET Entity Data Model to your project. Select Empty Model when prompted to choose the model contents. Click Finish. This will create an empty design surface.
2. Right-click the design surface and select Add ➤ Entity. Name the new entity User and click OK.
3. Right-click the new entity and add a scalar property for the UserName.
4. Right-click the design surface and select Add ➤ Entity. Name the new entity PasswordHistory and click OK.
5. Right-click the new entity and add a scalar property for the LastLogin. Right-click the LastLogin property and change its type to DateTime.
6. Right-click the User entity and select Add ➤ Association. To create a foreign key association, check the Add foreign key properties to the 'PasswordHistory' entity check box. TO create an independent association, uncheck this box.
7. Right-click the design surface and select Generate Model from Database. Select a database connection and complete the remainder of the wizard. This will generate the storage and mapping layers of the model and produce a script to generate the database for the model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

If you choose to create a foreign key association, the model should look like the one shown in Figure 6-38. If you choose to create an independent association, the model should look like the one shown in Figure 6-39.

Figure 6-38. *A foreign key association between User and PasswordHistory*

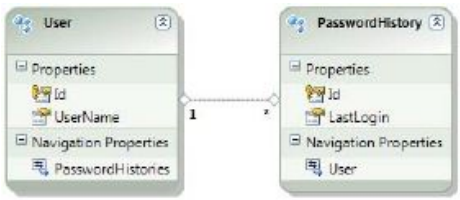


Figure 6-39. *An independent association between User and PasswordHistory*

How It Works

With a foreign key association, the foreign key is exposed as a property in the dependent entity. Exposing the foreign key allows many aspects of the association to be managed with the same code that manages the other property values. This is particularly helpful in disconnected scenarios as we will see in Chapter 9. Foreign key associations are the default in Entity Framework.

For independent associations, the foreign keys are not exposed as properties. This makes the modeling at the conceptual layer somewhat cleaner because there is no noise introduced concerning the details of the association implementation. In the early versions of Entity Framework, only independent associations where supported.

6-16. Changing an Independent Association into a Foreign Key Association

Problem

You have a model that uses an independent association and you want to change it to a foreign key association.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have a model like the one shown in Figure 6-40.

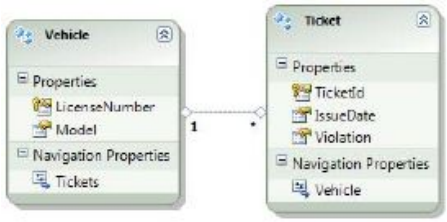


Figure 6-40. *A model for vehicles and tickets using an independent association*

To change the association from an independent association to a foreign key association, do the following:

1. Right-click the Ticket entity and select Add ➤ Scalar Property. Rename the property LicenseNumber.
2. View the Mapping Details window for the association. Remove the mapping to the Ticket table by selecting <Delete> from the Maps to Ticket control.
3. Right-click the association and view the properties. Click in the button in the Referential Constraint control. In the dialog box select the Vehicle entity in the Principal dropdown control. The Principal Key and the Dependent Property should both be set to LicenseNumber as shown in Figure 6-41.
4. View the Mapping Details window for the Ticket entity. Map the LicenseNumber column to the LicenseNumber property as shown in Figure 6-42.

The final model is shown in Figure 6-43.

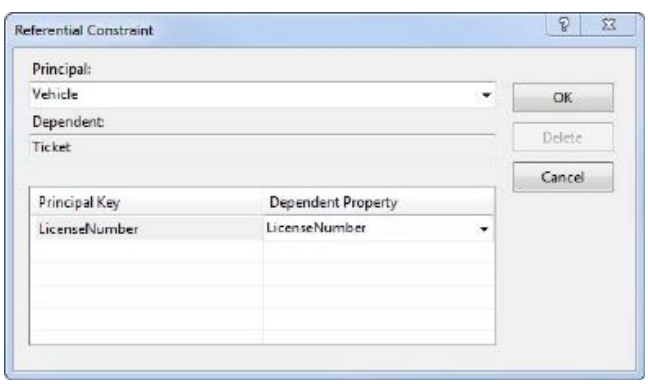


Figure 6-41. *Creating the referential constraint for the foreign key association*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

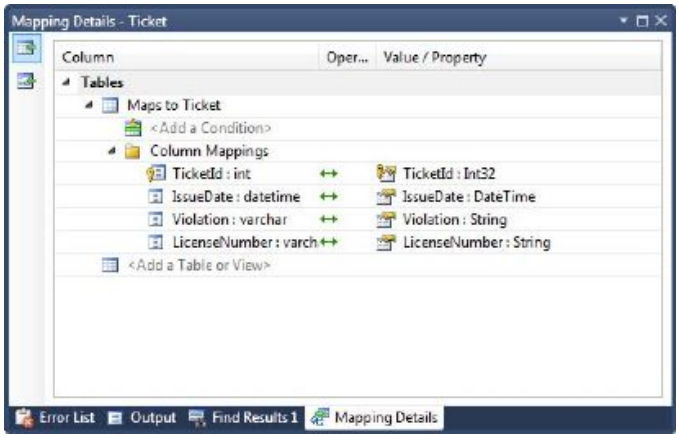


Figure 6-42. *Mapping the LicenseNumber column to the LicenseNumber property for the Ticket entity*

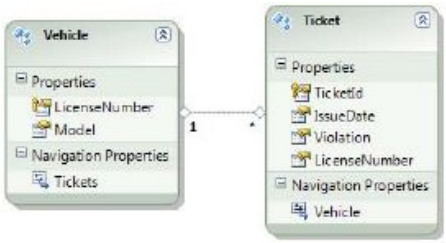


Figure 6-43. *The model with the independent association changed to a foreign key association*

How It Works

When you change an independent association into a foreign key association, most of your existing code will continue to work. You will find it easier now to associate two entities by simply setting the exposed foreign key to the appropriate value. To change a relationship with an independent association, you need to create a new instance of EntityKey and set the entity's xxxReference.EntityKey to this new instance. With a foreign key association, you simply set the exposed foreign key property to the key value.

Foreign key associations are not currently supported for many-to-many associations because these associations must be mapped to the underlying link table. A future version of Entity Framework may support foreign key associations along with payloads for many-to-many associations.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 7

Working with Object Services

This chapter contains a rather eclectic collection of recipes that provide practical solutions to common problems in real-world applications. We build our applications to tolerate changes in deployment environments and make our applications flexible enough so that few if any configuration details need to be hard-coded. The first three recipes provide you with tools to meet these challenges.

The remaining recipes cover topics such as Entity Framework's Pluralization Service, using the edmgen.exe utility, working with identifying relationships, and retrieving objects from an object context.

7-1. Dynamically Building a Connection String

Problem

You want to dynamically build the connection string for your application.

Solution

Many real-world applications start out on a developer's desktop; move through one or more testing, integration, and staging environments; and finally end up in a production deployment. You want to dynamically configure the application's connection string depending on the current environment.

To dynamically build the connection string for your application, follow the pattern in Listing 7-1.

Listing 7-1. Dynamically building a connection string

```
public static class ConnectionStringManager
{
    public static string EFConnection = GetConnection();

    private static string GetConnection()
    {
        var sqlBuilder = new SqlConnectionStringBuilder();

        // figure out the environment
        // strings here should come from a config file
        string myHost = Dns.GetHostName();
        if (myHost.ToLower().Contains("test"))
            sqlBuilder.DataSource = @"TestSql01";
        else if (myHost.ToLower().Contains("staging"))
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            sqlBuilder.DataSource = @"StagingSql01";
        else if (myHost.ToLower().Contains("prod"))
            sqlBuilder.DataSource = @"ProdSql01";
        else
            sqlBuilder.DataSource = @"localhost";

        // fill in the rest
        sqlBuilder.InitialCatalog = "EFRecipes";
        sqlBuilder.IntegratedSecurity = true;
        sqlBuilder.MultipleActiveResultSets = true;

        var eBuilder = new EntityConnectionStringBuilder();
        eBuilder.Provider = "System.Data.SqlClient";
        eBuilder.Metadata =
              "res://*/Recipe1.csdl|res://*/Recipe1.ssdl|res://*/Recipe1.msl";
        eBuilder.ProviderConnectionString = sqlBuilder.ToString();
        return eBuilder.ToString();
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.Connection.ConnectionString = ConnectionStringManager.EFConnection;
    }
}
```

How It Works

When you add an ADO.NET Entity Data Model to your project, Entity Framework adds an entry to the <connectionStrings> section in your project's .config file. At runtime, the constructor for the object context is passed the key for this configuration entry (EFRecipesEntities for the recipes in this book). Given this key, the object context uses the connection string found in the .config file.

To dynamically create the connection string based on the environment in which our application is deployed, we created the ConnectionStringManager class (refer to Listing 7-1). In the **GetConnection()** method, we check the name of the machine the application is on and use it to determine the target database server. To keep things simple, we hard-coded the names of machines here (you would probably want to put them in a .config file). To use our ConnectionStringManager, we implemented the **OnContextCreated()** partial method inside EFRecipesEntities partial class.

In our implementation of the **OnContextCreated()** partial method, we get the statically built connection string from the ConnectionStringManager. The object context will use this connection string to connect to our database server. You don't need to change anything else in your application. Each time you get a new instance of your object context, the **OnContextCreated()** method will get the connection string created when the static ConnectionStringManager class was created.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

7-2. Reading a Model from a Database

Problem

You want to read the CSDL, MSL, and SSDL definitions for your model from a database table.

Solution

Suppose that you have a model like the one in Figure 7-1.

Licensed by Matt Allen 2175280

Figure 7-1. *A model with a Customer entity*

Our model has just one entity: Customer. The conceptual layer (CSDL), mapping layer (MSL), and storage layer (SSDL) definitions are typically found in the .edmx file in your project. We want to read these definitions from a database. To read these definitions from a database, do the following:

1. Right-click the design surface and view the Properties. Change the Code Generation Strategy to None. We'll use POCO for our Customer class. See Chapter 8 for more recipes on using POCO.
2. Create the table shown in Figure 7-2. This table will hold the definitions for our project.
3. Right-click the design surface and view the Properties. Change the Metadata Artifact Processing to Copy to Output Directory. Rebuild your project. The build process will create three files in the output directory: Recipe2.ssdl, Recipe2.csdl, and Recipe2.msl.
4. Insert the contents of these files into the Definitions table in the corresponding columns. Use 1 for the Id column.
5. Follow the pattern in Listing 7-2 to read the metadata from the Definitions table and create a MetadataWorkspace that our application will use.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Definitions (Chapter7)

Column Name	Data Type	Allow Nulls
Id	int	
SSDL	xml	
CSDL	xml	
MSL	xml	

Figure 7-2. *The Definitions table holds the definitions for our SSDL, CSDL, and MSL. Note that the column data types for the definitions are XML.*

Listing 7-2. Reading the metadata from the Definitions table

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data.Metadata.Edm;
using System.Data.SqlClient;
using System.Data.EntityClient;
using System.Xml;
using System.Data.Mapping;
using System.Data.Objects;

namespace Recipe2
{
    class Program
    {
        static void Main(string[] args)
        {
            RunExample();
        }

        static void RunExample()
        {
            using (var context = ContextFactory.CreateContext())
            {
                context.Customers.AddObject(
                      new Customer { Name = "Jill Nickels" });
                context.Customers.AddObject(
                      new Customer { Name = "Robert Cole" });
                context.SaveChanges();
            }

            using (var context = ContextFactory.CreateContext())
            {
                Console.WriteLine("Customers");
                Console.WriteLine("---------");
                foreach (var customer in context.Customers)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            {
                Console.WriteLine("{0}", customer.Name);
            }
        }
    }
}

public class Customer
{
    public virtual int CustomerId { get; set; }
    public virtual string Name { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    private ObjectSet<Customer> customers;
    public EFRecipesEntities(EntityConnection cn)
        : base(cn)
    {
    }

    public ObjectSet<Customer> Customers
    {
        get
        {
            return customers ?? (customers = CreateObjectSet<Customer>());
        }
    }
}

public static class ContextFactory
{
    static string connString = @"Data Source=localhost;
       Initial Catalog=EFRecipes;Integrated Security=True;";
    private static MetadataWorkspace workspace = CreateWorkSpace();

    public static EFRecipesEntities CreateContext()
    {
        var conn = new EntityConnection(workspace,
                        new SqlConnection(connString));
        return new EFRecipesEntities(conn);
    }

    private static MetadataWorkspace CreateWorkSpace()
    {
        string sql = @"select csdl,msl,ssdl from Chapter7.Definitions";
        XmlReader csdlReader = null;
        XmlReader mslReader = null;
        XmlReader ssdlReader = null;

        using (var cn = new SqlConnection(connString))
        {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                using (var cmd = new SqlCommand(sql, cn))
                {
                    cn.Open();
                    var reader = cmd.ExecuteReader();
                    if (reader.Read())
                    {
                        csdlReader = reader.GetSqlXml(0).CreateReader();
                        mslReader = reader.GetSqlXml(1).CreateReader();
                        ssdlReader = reader.GetSqlXml(2).CreateReader();
                    }
                }
            }

            var workspace = new MetadataWorkspace();
            var edmCollection = new EdmItemCollection(new XmlReader[]
                                                { csdlReader });
            var ssdlCollection = new StoreItemCollection(new XmlReader[]
                                                { ssdlReader });
            var mappingCollection = new StorageMappingItemCollection(
                edmCollection, ssdlCollection, new XmlReader[] { mslReader });

            workspace.RegisterItemCollection(edmCollection);
            workspace.RegisterItemCollection(ssdlCollection);
            workspace.RegisterItemCollection(mappingCollection);
            return workspace;
        }
    }
}
```

The following is the output of the code in Listing 7-2:

```
Customers

---------

Jill Nickels

Robert Cole
```

How It Works

The first part of the code in Listing 7-2 should be very familiar to you by now. We use Entity Framework to create a new context, create a few entities, and call **SaveChanges()** to persist the entities to the database. To retrieve the entities, we iterate through the collection and display each on the console. The only difference in this part is the call to **ContextFactory.CreateContext()**. Normally, we would just use the **new** operator to get a new instance of our EFRecipesEntities context.

We've created the ContextFactory to create our context from the model metadata stored not in the .edmx file, but in a table in a database. We do this in the **CreateContext()** method. The **CreateContext()**

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

method creates a new EntityConnection based on two things: a workspace that we create with the `CreateWorkSpace()` method and a SQL connection string. The real work happens in how we create the workspace in the `CreateWorkSpace()` method.

The `CreateWorkSpace()` method opens a connection to the database where our metadata is stored. We construct a SQL statement that reads the one row from the Definitions table (refer to Figure 7-2) that holds our definitions for the conceptual layer, storage layer, and mapping layer. We read these definitions with XmlReaders. With these definitions, we create an instance of a MetadataWorkspace. A MetadataWorkspace is an in-memory representation of a model. Typically, this workspace is created by the default plumbing in Entity Framework from your .edmx file. In this recipe, we create this workspace from the definitions in a database. There are other ways to create this workspace including using embedded resources and an emerging perspective called Code First.

The code in Listing 7-2 uses Plain Old CLR Objects, also known as POCO, for our Customer entity. We cover POCO extensively in Chapter 8, but here we use POCO to simplify the code. With POCO, we don't use the classes generated by Entity Framework. Instead, we use our own classes that have no particular dependence on Entity Framework. In Listing 7-2, we created our own definition of the Customer entity in the Customer class. We also created our own object context: EFRecipesEntities. Our context, of course, does have a dependence on Entity Framework because it derives from ObjectContext.

7-3. Deploying a Model

Problem

You want to know the various options for deploying a model.

Solution

When you add a new ADO.NET Entity Data Model to your project, Entity Framework sets the Build Action property for the .edmx file to Entity Deploy. Additionally, the Metadata Artifact Processing property of the model is set to Embed in Output Assembly. When you build your project, the Entity Deploy action extracts three sections from the .edmx file into three separate files. The CSDL section is extracted into the Model.csdl file. The MSL section is extracted into the Model.msl file. The SSDL section is extracted into the Model.ssdl file. With the Embed in Output Assembly, these three files get embedded into the assembly as resources.

Changing the Metadata Artifact Processing property to Copy to Output Directory causes the three Model.* files to be copied to the same directory as the resulting assembly. The files are not embedded as a resource.

How It Works

The .edmx file contains all three model layers: conceptual, mapping, and storage. The file also contains additional data used by the designer to manage the design surface. At runtime, Entity Framework uses each of the layers separately. The .edmx file is just a convenient container for the design time user experience. The deployment of a model depends on model layers either embedded in the assembly, stored in files, or, as we saw in Recipe 7-2, retrieved from another source and used to complete a MetadataWorkspace.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

If your Metadata Artifact Processing property is set to Embed in Output Assembly, you will notice that the connection string in your App.config or web.config file, includes a **metadata** tag, which looks something like the following:

```
metadata=res://*/Recipe3.csdl|res://*/Recipe3.ssdl|res://*/Recipe3.msl;
```

This notation indicates a search path for each of the model layers embedded in the assembly. If you change the Metadata Artifact Processing property to Copy to Output Directory, you will see the connection string change to something like this:

```
metadata=.\Recipe3.csdl|.\Recipe3.ssdl|.\Recipe3.msl;
```

This notation indicates a file path to each of the model layers.

When embedding the model layers as resources in an assembly, you are not restricted by the connection string syntax to referencing only the executing assembly. Table 7-1 illustrates some of the possible constructions you can use to reference the embedded model layers in other assemblies.

Table 7-1. *Connection String Syntax for Loading Model Layers*

Syntax	Meaning
res://myassembly/file.ssdl	Loads the SSDL from myassembly
res://myassembly/	Loads the SSDL, CSDL, and MSL from myassembly
res://*/file.ssdl	Loads the SSDL from all assemblies in the AppDomain
res://*/	Loads the SSDL, CSDL, and MSL from all assemblies

7-4. Using the Pluralization Service

Problem

You want to use Entity Framework's Pluralization Service when you import table from a database.

Solution

Suppose that you have a database with the tables shown in Figure 7-3.

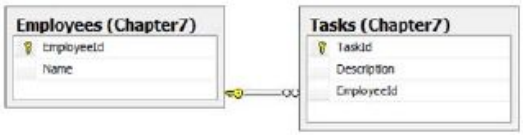


Figure 7-3. *Employees and Tasks tables in our database*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Notice that the tables in Figure 7-3 take the plural form. This is common in many databases. Some DBAs believe that all table names should be plural; other DBAs believe just the opposite. And, of course, there are a few who don't seem to follow any particular view and mix things up. Depending on your perspective, you may want to use the singular form of the table names for your model's entities. Entity Framework provides a Pluralization Service that can automatically generate the singular form of a table name to use as the corresponding entity name.

To use the Pluralization Service when importing your tables, check the Pluralize or singularize generated object names box in the last step of the Entity Data Model Wizard (see Figure 7-4). By default, this box is checked.

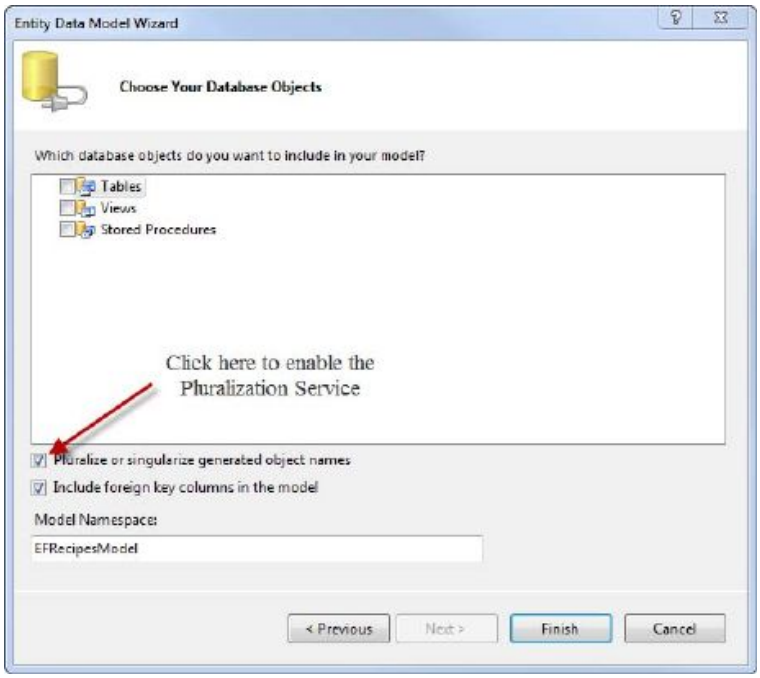


Figure 7-4. *Enabling the Pluralization Service*

Figure 7-5 shows a model created when we import the table in Figure 7-3 without the Pluralization Service enabled. Notice that entity names are taken directly from the table names and retain the plural form. Figure 7-6 shows the same tables imported with the Pluralization Service enabled. These entities use the singular forms of the table names.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 7-5. *The model created from the tables in Figure 7-3 without the Pluralization Service*

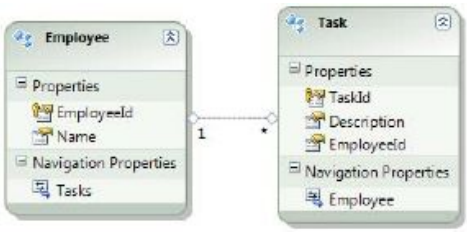


Figure 7-6. *The model created from the tables in Figure 7-3 with the Pluralization Service*

How It Works

Most developers prefer the entity names in the model in Figure 7-6. (Look at the names in boldface at the top of each entity). Not only are the entity names singular but the Employee navigation property in the Task entity also makes more sense than the Employees navigation property in the Tasks entity in Figure 7-5. In both cases, this navigation property is an EntityReference, not a collection. The plural form in Figure 7-5 seems somewhat confusing.

If our table names were singular to start with, the Pluralization Service would correctly pluralize the collection-based navigation properties and pluralize the underlying entity set names. This takes cares of the other half of the DBA community that use singular names for tables.

You can set the default on/off state of the Pluralization Service for new entities in your model by changing the Pluralize New Objects property. When you add new entities to your model, this setting will change the default on/off state for the Pluralization Service.

You can use the Pluralization Service outside of the context of Entity Framework. This service is available in the System.Data.Entity.Design namespace. To add a reference to the System.Data.Entity.Design.dll, you will need to change your project's Target framework from the default .NET Framework 4 Client Profile to the more expansive, .NET Framework 4. This setting is changed in the properties of the project. The code in Listing 7-3 demonstrates using the Pluralization Service to pluralize and singularize the words *person* and *people*.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 7-3. Using the Pluralization Service

```
var service = PluralizationService.CreateService(new CultureInfo("en-US"));
string person = "Person";
string people = "People";
Console.WriteLine("The plural of {0} is {1}", person,
                  service.Pluralize(person));
Console.WriteLine("The singular of {0} is {1}", people,
                  service.Singularize(people));
```

The following is the output of the code in Listing 7-3:

```
The plural of Person is People
The singular of People is Person
```

7-5. Retrieving Entities from the Object State Manager

Problem

You want to create an extension method that retrieves entities from the object state manager.

Solution

Suppose you have a model like the one in Figure 7-7.

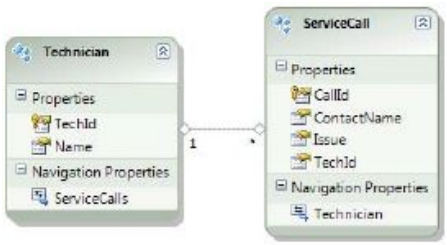


Figure 7-7. *Our model with technicians and their service calls*

In this model, each technician has service calls that include the contact name and issue for the call. You want to create an extension method that retrieves all entities in the model that are in the Added, Modified, or Unchanged state. To do this, follow the pattern in Listing 7-4.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 7-4. Creating an extension method that retrieves all the entities in the Added, Modified, or Unchanged state

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var tech1 = new Technician { Name = "Julie Kerns" };
            var tech2 = new Technician { Name = "Robert Allison" };
            context.ServiceCalls.AddObject(new ServiceCall {
                  ContactName = "Robin Rosen",
                  Issue = "Can't get satellite signal.",
                  Technician = tech1 });
            context.ServiceCalls.AddObject(new ServiceCall {
                  ContactName = "Phillip Marlowe",
                  Issue = "Channel not available",
                  Technician = tech2 });

            // now get the entities we've added
            foreach (var tech in
                     context.ObjectStateManager.GetEntities<Technician>())
            {
                Console.WriteLine("Technician: {0}", tech.Name);
                foreach (var call in tech.ServiceCalls)
                {
                    Console.WriteLine("\tService Call: Contact {0} about {1}",
                                      call.ContactName, call.Issue);
                }
            }
        }
    }
}

public static class StateManagerExtensions
{
    public static IEnumerable<T> GetEntities<T>(this ObjectStateManager manager)
    {
        var entities = manager
                 .GetObjectStateEntries(~EntityState.Detached)
                 .Where(entry => !entry.IsRelationship && entry.Entity != null)
                 .Select(entry => entry.Entity).OfType<T>();
        return entities;
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following is the output of the code in Listing 7-4:

```
Technician: Julie Kerns

        Service Call: Contact Robin Rosen about Can't get satellite signal.

Technician: Robert Allison

        Service Call: Contact Phillip Marlowe about Channel not available
```

How It Works

Licensed by mallen@iq 2175280

In Listing 7-4, we implemented the **GetEntities<T>()** extension method to retrieve all the entities in the object context that are in the Added, Modified, or Unchanged state. Because this may be a common activity in your application, it makes sense to implement this just once in an extension method. In the implementation of the **GetEntities<T>()** method, we call the **GetObjectStateEntries()** method passing in the **~EntityState.Detached** expression. The method returns all entries that are not in the Detached state. From these, we filter out relationships and null entries. From the remaining entries, we select only those of the given type.

There are some important scenarios in which you might want to implement a method like **GetEntities<T>()**. For example, in the **SavingChanges** event, you may want to validate entities that are about to be inserted, modified, or deleted.

It is important to note that when you add or delete entities from the object context, these changes are not reflected in results of queries against the object context. These queries represent entities as they exist in the database, not what currently exist in the object context.

In our implementation of **GetEntities<T>()**, we filtered out relationship entries in the object state manager. Relationships are first-class objects in Entity Framework, and entries are created in the object state manager for relationships.

7-6. Generating a Model from the Command Line

Problem

You want to generate a model from the command line.

Solution

To generate a model for a given database from the command line, use the edmgen.exe program. To access the Visual Studio Command Prompt, click Visual Studio 2010 Command Prompt under Microsoft Visual Studio 2010 from the Start menu.

The Microsoft documentation for the edmgen command provides a complete list of the command line options. The edmgen command supports a lot of useful command line options. The following command, for example, will generate a model from all of the tables in the given Test database:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
edmgen /mode:FullGeneration /project:Test /provider:"System.Data.SqlClient"
/c:"server=localhost;integrated security=true;database=Test;"
```

Other /mode options are available. One that can be particularly useful in a continuous integration build process is **/mode:ValidateArtifacts**. With this option, one or more of the generated layers are validated. You need to use one or both of the **/inssdl** or **/incsdl** options. If you are validating the mapping layer, all three layers must be specified.

You can use one of the /out options to specify the name of the generated file for specific model layers. For example, using **/outcsdl:MyProject.csdl** will create the conceptual layer definitions in a file named MyProject.csdl. There are similar options for the other layers.

How It Works

The edmgen command provides a convenient way to automate some of the build processes and is a useful tool for pregenerating views and generating separate files for the model layers. One restriction of edmgen is that it does not provide a way to generate a model based on a subset of the tables in a database.

7-7. Working with Dependent Entities in an Identifying Relationship

Problem

You want to insert, update, and delete a dependent entity in an identifying relationship.

Solution

Suppose you have a model like the one in Figure 7-8. The LineItem's entity key is a composite key comprised of InvoiceNumber and ItemNumber. InvoiceNumber is also a foreign key to the Invoice entity.

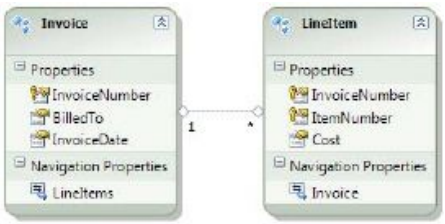


Figure 7-8. *Invoice and LineItem in an identifying relationship because of the composite entity key in the LineItem entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When one of the properties of an entity key is both the primary key and the foreign key, the entity is said to be participating in an identifying relationship. In our model, LineItem's entity key, its identity, is also a foreign key to the Invoice entity. The LineItem entity is referred to as the *dependent entity* while Invoice is the *principal entity*.

There is a subtle difference in how Entity Framework handles the deletion of dependent entities in an identifying relationship. Because the dependent entity cannot exist without participating in the relationship, simply removing the dependent entity from the principal's collection will result in Entity Framework marking the dependent entity for deletion. Additionally, deleting the principal entity will also mark the dependent for deletion. This is reminiscent of the cascading deletes common in database systems. Of course, Entity Framework allows you to explicitly delete the dependent entity. The code in Listing 7-5 demonstrates all three of these scenarios.

Listing 7-5. Deleting the dependent entity

```
static void Main(string[] args)
{
    RunExample();
}

static void RunExample()
{
    using (var context = new EFRecipesEntities())
    {

        var invoice1 = new Invoice { BilledTo = "Julie Kerns",
                         InvoiceDate = DateTime.Parse("3/19/2010") };
        var invoice2 = new Invoice { BilledTo = "Jim Stevens",
                         InvoiceDate = DateTime.Parse("3/21/2010") };
        context.LineItems.AddObject(new LineItem { Cost = 99.29M,
                                   Invoice = invoice1 });
        context.LineItems.AddObject(new LineItem { Cost = 29.95M,
                                   Invoice = invoice1 });
        context.LineItems.AddObject(new LineItem { Cost = 109.95M,
                                   Invoice = invoice2 });
        context.SaveChanges();

        // display the line items
        Console.WriteLine("Original set of line items...");
        DisplayLineItems();

        // remove a line item from invoice 1's collection
        var item = invoice1.LineItems.ToList().First();
        invoice1.LineItems.Remove(item);
        context.SaveChanges();
        Console.WriteLine("\nAfter removing a line item from an invoice...");
        DisplayLineItems();

        // remove invoice2
        context.DeleteObject(invoice2);
        context.SaveChanges();
        Console.WriteLine("\nAfter removing an invoice...");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        DisplayLineItems();

        // remove a single line item
        context.DeleteObject(invoice1.LineItems.First());
        context.SaveChanges();
        Console.WriteLine("\nAfter removing a line item...");
        DisplayLineItems();
    }
}

static void DisplayLineItems()
{
    bool found = false;
    using (var context = new EFRecipesEntities())
    {
        foreach (var lineitem in context.LineItems)
        {
            Console.WriteLine("Line item: Cost {0}",
                              lineitem.Cost.ToString("C"));
            found = true;
        }
    }
    if (!found)
        Console.WriteLine("No line items found!");
}
```

The following is the output of the code in Listing 7-5:

```
Original set of line items...

Line item: Cost $99.29

Line item: Cost $29.95

Line item: Cost $109.95



After removing a line item from an invoice...

Line item: Cost $29.95

Line item: Cost $109.95



After removing an invoice...

Line item: Cost $29.95
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
After removing a line item...

No line items found!
```

How It Works

The code in Listing 7-5 deletes line items in three ways. First, it deletes a line item from an invoice's collection. Because a line item is dependent on the invoice for its identity, Entity Framework marks the referenced line item for deletion. Next, it deletes an invoice. Entity Framework marks all the dependent line items for deletion. Finally, the code deletes the last remaining line item directly by calling `DeleteObject()`.

You can modify all the properties of a dependent entity except for properties that participate in the identifying relationship. In our model, we can modify the Cost property in a line item, but we can't change the Invoice navigation property.

When a principal object in an identifying relationship is saved to the database, the key that is generated at the database (for store-generated values) is written to the principal entity and to all its dependent entities. This ensures that all are synchronized in the object context.

The subtle difference in Entity Framework's treatment of a deleted relationship between two entities in an identifying relationship and two entities in any other relationship is worth noting. For other types of relationships, Entity Framework does not mark an entity for deletion if the entity is removed from the collection of another entity. For an identifying relationship, Entity Framework does mark the dependent entity for deletion.

7-8. Inserting Entities Using an Object Context

Problem

You want to insert entities in your model to the database using an object context.

Solution

Suppose you have a model like the one in Figure 7-9.

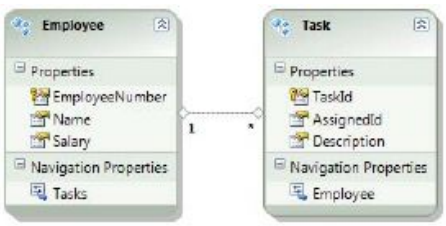


Figure 7-9. *A model with employees and their tasks*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The model in Figure 7-9 represents employees and their tasks. You want to insert new employees and their tasks into the underlying database. To insert an Employee, create a new instance of Employee and call the **AddObject()** method available on the Employees entity set in the context. To add a Task for an employee, create a new instance of Task and add it to the Tasks collection of the employee. You must also call **AddObject()** to add either the employee or the task to the object context. To persist the changes to the database, call the **SaveChanges()** method.

The code in Listing 7-6 demonstrates using **AddObject()** to add new objects to the object context and persist them to the database with **SaveChanges()**.

Listing 7-6. Inserting new entities into the database

```
using (var context = new EFRecipesEntities())
{
    var employee1 = new Employee {EmployeeNumber = 629,
                                  Name = "Robin Rosen", Salary = 106000M };
    var employee2 = new Employee {EmployeeNumber = 147,
                                  Name = "Bill Moore", Salary = 62500M };
    var task1 = new Task { Description = "Report 3rd Qtr Accounting" };
    var task2 = new Task { Description = "Forecast 4th Qtr Sales" };
    var task3 = new Task { Description = "Prepare Sales Tax Report" };

    // use AddObject() on the Employees entity set
    context.Employees.AddObject(employee1);

    // add two new tasks to the employee1's tasks
    employee1.Tasks.Add(task1);
    employee1.Tasks.Add(task2);

    // add a task to the employee and use
    // AddObject() to add the task to the object context
    employee2.Tasks.Add(task3);
    context.Tasks.AddObject(task3);

    // persist all of these to the database
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var employee in context.Employees)
    {
        Console.WriteLine("Employee: {0}'s Tasks", employee.Name);
        foreach (var task in employee.Tasks)
        {
            Console.WriteLine("\t{0}", task.Description);
        }
    }
}
```

The following is the output of the code in Listing 7-6:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Employee: Bill Moore's Tasks

        Prepare Sales Tax Report

Employee: Robin Rosen's Tasks

        Report 3rd Qtr Accounting

        Forecast 4th Qtr Sales
```

How It Works

In Listing 7-6, we used the **AddObject()** method available on the Employees and Tasks entity sets to add entities to the object context. An **AddObject()** method is also available on the object context. This second version of **AddObject()** exists for largely historic reasons. Most new applications use the **AddObject()** method on the entity set.

When you add an entity to the object context, Entity Framework creates a temporary entity key for the newly added entity. Entity Framework uses this temporary key to uniquely identify the entity. This temporary key is replaced by a real key after the object is persisted to the database. If saving two entities to the database results in both entities being assigned the same entity key, Entity Framework will throw an exception. This can happen if the keys are assigned the same value by the client or by some store-generating process.

For foreign key associations, you can assign the foreign key property of an entity the value of the entity key of a related entity. Although temporary keys are involved, Entity Framework will fix up the keys and relationships correctly when the entities are saved to the database.

You can also use the **Attach()** method to add an entity to an object context. This is a two-step process. First, call **Attach()** with the entity. This adds it to the object context, but the object state manager initially marks the entity as Unchanged. Calling **SaveChanges()** at this point will not save the entity to the database. The second step is to call the object state manager's **ChangeObjectState()** method passing in the new state: **EntityState.Added**. Calling **SaveChanges()** at this point will save the new entity to the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 8

Plain Old CLR Objects

Objects should not know how to save themselves, or load themselves, or filter themselves. That's a familiar mantra in software development and especially in Domain Driven Development. There is a good bit of wisdom in the mantra. Having persistence knowledge bound too tightly to our domain objects complicates testing, refactoring, and reuse. The classes generated by Entity Framework for our model entities are heavily dependent on the plumbing of Entity Framework. For some developers, these classes know too much about the persistence mechanism and are too closely tied to the concerns of models and mapping. There is another option.

The Entity Framework also supports using your own classes for the entities in the model. The term *Plain Old CLR Objects*, often referred to simply as *POCO*, isn't meant to imply that your classes are either plain or old, but simply that they don't contain any reference at all to specialized frameworks. They don't need to derive from third-party code. They don't need to implement any special interface. And they don't need to live in any special assembly or namespace. You implement your domain objects however you see fit and tie them to the model with a custom object context. With that, you are all set to leverage all the power of Entity Framework and follow just about any architectural pattern you choose.

This chapter covers a wide variety of recipes specific to POCO. The first recipe shows you the basics of using POCO. The remaining recipes focus on loading entities and keeping Entity Framework in sync with state of your objects.

In this chapter, we've purposefully focused on writing most of the POCO-related code by hand to demonstrate how things work. Much of the work of building the POCO plumbing goes away if you use the POCO T4 template available from the ADO.NET development team at Microsoft.

8-1. Using POCO

Problem

You want to use Plain Old CLR Objects (POCO) in your application.

Solution

Let's say you have a data model like the one shown in Figure 8-1.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

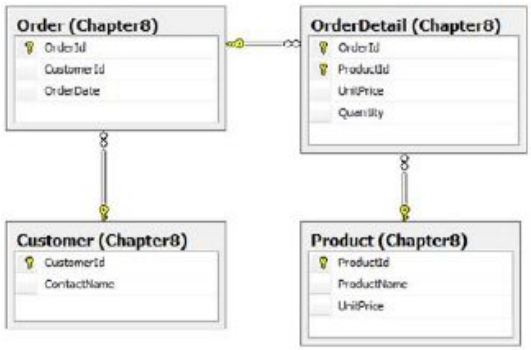


Figure 8-1. *A database model for customers and their orders*

To create an Entity Framework model based on the database tables in Figure 8-1 and using your own classes representing an Order, OrderDetail, Customer, and Product, do the following:

1. Right-click your project and select Add ➤New Item.
2. From the Visual C# Items Data templates, select ADO.NET Entity Data Model.
3. Select Generate from database to create the model from our existing tables.
4. Select the Order, OrderDetail, Customer, and Product tables; and click Next.
5. In the generated model, the Product entity has an OrderDetails navigation property for all the order details associated with this product. This is unnecessary here, so delete this navigation property. The completed model is shown in Figure 8-2.

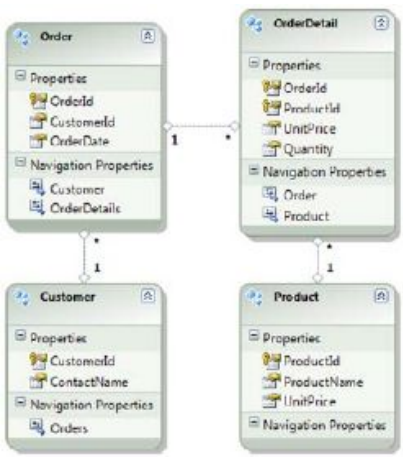


Figure 8-2. *The model for our customers' orders*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

6. We won't be using generated classes for our entities, so turn off code generation for the model. Right-click the design surface and select Properties. Change the Code Generation Strategy to None.

7. Create a class for each of the entities in our model. Make sure you use the exact same name for the class and each property as we have in the model. Use **ISet<T>** as the type for each of the navigation properties. The code in Listing 8-1 shows the classes for our model.

Listing 8-1. The plain old CLR classes for our model

```
public class Customer
{
    public int CustomerId { get; set; }
    public string ContactName { get; set; }
    public ISet<Order> Orders {get; set;}
    public Customer()
    {
        this.Orders = new HashSet<Order>();
    }
}

public class Order
{
    public int OrderId {get; set;}
    public int CustomerId {get; set;}
    public DateTime OrderDate {get; set;}
    public Customer Customer {get; set;}
    public ISet<OrderDetail> OrderDetails {get; set;}
    public Order()
    {
        this.OrderDetails = new HashSet<OrderDetail>();
    }
}

public class OrderDetail
{
    public int OrderId {get; set;}
    public int ProductId {get; set;}
    public decimal UnitPrice {get; set;}
    public int Quantity {get; set;}
    public Order Order {get; set;}
    public Product Product {get; set;}
}

public class Product
{
    public int ProductId {get; set;}
    public string ProductName {get; set;}
    public decimal UnitPrice {get; set;}
}
```

Licensed by Matt Allen 6175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Notice that there is no association from Product to OrderDetail because we removed that navigation property in the designer.

8. To use our POCO classes, we need to create another class that is derived from **ObjectContext**. This class will expose an **ObjectSet<T>** for each of the entities in our model. The code in Listing 8-2 illustrates how we might define this class.

Listing 8-2. Creating an ObjectContext for our model

```
public class EFRecipesEntities : ObjectContext
{
    private ObjectSet<Customer> _customers;
    private ObjectSet<Order> _orders;
    private ObjectSet<OrderDetail> _orderdetails;
    private ObjectSet<Product> _products;

    public EFRecipesEntities() :
      base("name=EFRecipesEntities","EFRecipesEntities")
    {
        _orders = CreateObjectSet<Order>();
        _orderdetails = CreateObjectSet<OrderDetail>();
        _products = CreateObjectSet<Product>();
    }

    public ObjectSet<Customer> Customers
    {
        get { return _customers ?? (_customers = CreateObjectSet<Customer>()); }
    }

    public ObjectSet<Order> Orders
    {
        get { return _orders; }
    }

    public ObjectSet<OrderDetail> OrderDetails
    {
        get { return _orderdetails; }
    }

    public ObjectSet<Product> Products
    {
        get { return _products; }
    }
}
```

This completes our model with our POCO classes. We can now use our model with our POCO classes just as we would use the model with the generated classes. The code in Listing 8-3 illustrates this.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 8-3. Using our POCO classes

```
using (var context = new EFRecipesEntities())
{
    var tea = new Product { ProductName = "Green Tea", UnitPrice = 1.09M };
    var coffee = new Product {ProductName = "Colombian Coffee",
                              UnitPrice = 2.15M};
    var customer = new Customer { ContactName = "Karen Marlowe" };
    var order1 = new Order { OrderDate = DateTime.Parse("4/19/10") };
    order1.OrderDetails.Add(new OrderDetail { Product = tea,
                              Quantity = 4, UnitPrice = 1.00M });
    order1.OrderDetails.Add(new OrderDetail { Product = coffee,
                              Quantity = 3, UnitPrice = 2.15M });
    customer.Orders.Add(order1);
    context.Customers.AddObject(customer);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var query = context.Customers.Include("Orders.OrderDetails.Product");
    foreach (var customer in query)
    {
        Console.WriteLine("Orders for {0}", customer.ContactName);
        foreach (var order in customer.Orders)
        {
            Console.WriteLine("--Order Date: {0}--",
                        order.OrderDate.ToShortDateString());
            foreach (var detail in order.OrderDetails)
            {
                Console.WriteLine(
                    "\t{0}, {1} units at {2} each, unit discount: {3}",
                    detail.Product.ProductName,
                    detail.Quantity.ToString(),
                    detail.UnitPrice.ToString("C"),
                    (detail.Product.UnitPrice - detail.UnitPrice).ToString("C"));
            }
        }
    }
}
```

The following is the output of the code in Listing 8-3:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Orders for Karen Marlowe
--Order Date: 4/19/2010--
        Green Tea, 4 units at $1.00 each, unit discount: $0.09
        Colombian Coffee, 3 units at $2.15 each, unit discount: $0.00
```

How It Works

To substitute our own classes for the ones typically generated by the Entity Framework from a model, we started off by setting the Code Generation Strategy property value to None. This keeps Entity Framework from generating code for our model and allows us to use our own classes, which don't derive from anything special and are "untainted" by any reference to Entity Framework.

Of course, we need to create a class corresponding to each of the entities in our model. As you can see from Listing 8-1, they are pretty simple and clean. We have to be careful to name the class exactly the same as the entity, and we have to have the corresponding properties. The Entity Framework is smart enough to find our classes almost anywhere they are defined in the AppDomain, so you don't need to put them in any special namespace or even in the same assembly as the model.

Of course, without code generation no object context is generated for us. To implement an object context that is specific to our model and our entities, we simply create a new class that derived from ObjectContext and provides properties of type `ObjectSet<T>` corresponding to each of the object sets in our context. You can choose to initialize each of these object sets in the constructor as we have for Product, Order, and OrderDetail. Or you can initialize them when accessed as we have for Customer.

Our EFRecipesEntities object context needs to be connected to the underlying database just like a generated object context. To do this, we pass two things to the base constructor. First, we pass in the connection string that was added by the designer to the config file. The second parameter is the default entity container name, which is usually the same name as the connection string.

The code in Listing 8-3 demonstrates inserting into and querying our model. Notice that we follow the same basic pattern when interacting with our POCO classes as we have previously with model-generated code.

By default, POCO classes do not support lazy loading or change tracking. However, with proxies, you can get both of these behaviors. We'll cover proxies in recipe 8-3.

8-2. Loading Related Entities With POCO

Problem

Using POCO, you want to explicitly load related entities.

Solution

Suppose you have a model like the one in Figure 8-3.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 8-3. *A model representing venues, their events, and the competitors in the events*

We're using POCO for our entities and we want to explicitly load the related entities (navigation properties). To do this we use the **LoadProperty()** method available on the object context. The code in Listing 8-4 illustrates using the **LoadProperty()** method.

*Listing 8-4. Using the **LoadProperty()** method to explicitly load navigation properties*

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipeEntities())
        {
            var venue = new Venue { Name = "City Center Hall" };
            var event1 = new Event { Name = "All Star Boxing" };
            event1.Competitors.Add(new Competitor { Name = "Big Joe Green" });
            event1.Competitors.Add(new Competitor { Name = "Terminator Tim" });
            venue.Events.Add(event1);
            context.Venues.AddObject(venue);
            context.SaveChanges();
        }

        using (var context = new EFRecipeEntities())
        {
            foreach (var venue in context.Venues)
            {
                Console.WriteLine("Venue: {0}", venue.Name);
                context.LoadProperty(venue, v => v.Events);
                foreach (var evt in venue.Events)
                {
                    Console.WriteLine("\tEvent: {0}", evt.Name);
                    Console.WriteLine("\t--- Competitors ---");
                    context.LoadProperty(evt, e => e.Competitors);
                    foreach (var competitor in evt.Competitors)
                    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                        Console.WriteLine("\t{0}", competitor.Name);
                    }
                }
            }
        }
    }
}

public class Venue
{
    public int VenueId { get; set; }
    public string Name { get; set; }
    public ICollection<Event> Events { get; set; }
    public Venue()
    {
        this.Events = new HashSet<Event>();
    }
}

public class Event
{
    public int EventId { get; set; }
    public string Name { get; set; }
    public int VenueId { get; set; }
    public Venue Venue { get; set; }
    public ICollection<Competitor> Competitors { get; set; }
    public Event()
    {
        this.Competitors = new HashSet<Competitor>();
    }
}

public class Competitor
{
    public int CompetitorId { get; set; }
    public string Name { get; set; }
    public int EventId { get; set; }
}

public class EFRecipeEntities : ObjectContext
{
    public EFRecipeEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private ObjectSet<Venue> venues;
    private ObjectSet<Event> events;
    private ObjectSet<Competitor> competitors;

    public ObjectSet<Venue> Venues
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        get { return venues ?? (venues = CreateObjectSet<Venue>());}
    }

    public ObjectSet<Event> Events
    {
        get { return events ?? (events = CreateObjectSet<Event>());}
    }

    public ObjectSet<Competitor> Competitors
    {
        get { return competitors ?? (competitors = CreateObjectSet<Competitor>());
    }
}
```

The following is the output of the code in Listing 8-4:

```
Venue: City Center Hall

        Event: All Star Boxing

        --- Competitors ---

        Big Joe Green

        Terminator Tim
```

How It Works

When we're using code generated by Entity Framework for our model, we use the **Load()** method available on the EntityReference and EntityCollection properties. For POCO, these methods are not available because the corresponding values for our properties are either instances of POCO classes or **ICollection<T>**.

To explicitly load a navigation property when we use POCO, we need to use the **LoadProperty()** method exposed on the object context. Recall that with POCO, we implement our own object context, but because it derived from **ObjectContext**, it comes with an implementation of the **LoadProperty()** method.

We need to explicitly load the navigation properties when we use POCO and are not using proxies. Proxies are generated when properties are marked as virtual. When proxies are generated, you can control lazy loading with the **ContextOptions.LazyLoadingEnabled** Boolean.

8-3. Lazy Loading With POCO

Problem

You are using plain old CLR objects and you want to lazy load related entities.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have a model like the one in Figure 8-4.

Figure 8-4. *A simple model for traffic tickets, the offending vehicles, and the details of the violation*

To enable lazy loading, we need to do two things. First, set **LazyLoadingEnabled** to **true** on the object context's **ContextOptions**. Next, mark the properties we want lazy loaded as **virtual**. This will cause proxy objects to be created for the entities which will handle the lazy loading. The code in Listing 8-5 illustrates this approach.

*Listing 8-5. Marking properties as **virtual** and setting **LazyLoadingEnable** to **true***

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var vh1 = new Vehicle { LicenseNo = "BR-549" };
            var t1 = new Ticket { IssueDate = DateTime.Parse("4/18/10") };
            var v1 = new Violation {
                         Description = "20 MPH over the speed limit",
                         Amount = 125M };
            var v2 = new Violation {
                         Description = "Broken tail light",
                         Amount = 50M };
            t1.Violations.Add(v1);
            t1.Violations.Add(v2);
            t1.Vehicle = vh1;
            context.Tickets.AddObject(t1);
            var vh2 = new Vehicle { LicenseNo = "XJY-902" };
            var t2 = new Ticket { IssueDate = DateTime.Parse("4/20/10") };
            var v3 = new Violation {
                         Description = "Parking in a no parking zone",
                         Amount = 35M };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            t2.Violations.Add(v3);
            t2.Vehicle = vh2;
            context.Tickets.AddObject(t2);
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            context.ContextOptions.LazyLoadingEnabled = true;
            foreach (var ticket in context.Tickets)
            {
                Console.WriteLine(" Ticket: {0}, Total Cost: {1}",
                  ticket.TicketId.ToString(),
                  ticket.Violations.Sum(v => v.Amount).ToString("C"));
                foreach (var violation in ticket.Violations)
                {
                    Console.WriteLine("\t{0}", violation.Description);
                }
            }
        }
    }
}

public class Ticket
{
    public int TicketId { get; set; }
    public int VehicleId { get; set; }
    public DateTime IssueDate { get; set; }
    public virtual Vehicle Vehicle {get; set;}
    public virtual ICollection<Violation> Violations {get; private set;}
    public Ticket()
    {
        this.Violations = new HashSet<Violation>();
    }
}

public class Vehicle
{
    public int VehicleId {get; set;}
    public string LicenseNo {get; set;}
}

public class Violation
{
    public int ViolationId { get; set; }
    public int TicketId { get; set; }
    public string Description { get; set; }
    public decimal Amount { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private ObjectSet<Ticket> tickets;
    private ObjectSet<Violation> violations;
    private ObjectSet<Vehicle> vehicles;

    public ObjectSet<Ticket> Tickets
    {
        get { return tickets ?? (tickets = CreateObjectSet<Ticket>()); }
    }

    public ObjectSet<Violation> Violations
    {
        get { return violations ?? (violations = CreateObjectSet<Violation>()); }
    }

    public ObjectSet<Vehicle> Vehicles
    {
        get { return vehicles ?? (vehicles = CreateObjectSet<Vehicle>()); }
    }
}
```

The following is the output of the code in Listing 8-5:

```
Ticket: 1, Total Cost: $175.00

        20 MPH over the speed limit

        Broken tail light

Ticket: 2, Total Cost: $35.00

        Parking in a no parking zone
```

How It Works

To get lazy loading, we first needed to enable it by setting the **LazyLoadingEnabled** property on the **ContextOptions** to **true**. This turns on lazy loading for the context.

Next, we let Entity Framework create proxy objects for our POCO objects by marking as **virtual** the navigation properties we want to lazy load. These proxy objects derive from our POCO objects and act in place of our objects.

There are two subtly different effects from marking properties as **virtual**. If we mark all the properties as **virtual**, we get proxies that provide both change tracking and lazy loading. The lazy loading, also, of course, depends on **LazyLoadingEnabled** being **true**. If we mark just the navigation

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

properties as **virtual**, the end result is that we get proxies that support just lazy loading, not change tracking.

8-4. POCO With Complex Type Properties

Problem

You want to use a complex type in your POCO entity.

Solution

Licensed by Matt Allen 2175280

Suppose your model looks like the one in Figure 8-5. In this model, the Name property is a complex type.

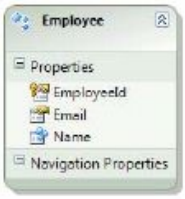


Figure 8-5. *A model for an employee. The Name property is a complex type composed of FirstName and LastName.*

Complex types are supported with POCO. Simply create a **class** for the complex type and use this class as the type for the property. Only classes are supported. There is no current support for **struct** as a complex type. The code in Listing 8-6 illustrates using the Name class for the complex type representing the employee's FirstName and LastName.

Listing 8-6. Using a complex type with POCO

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            context.Employees.AddObject(new Employee {
                   Name = new Name { FirstName = "Annie",
                                     LastName = "Oakley" },
                   Email = "aoakley@wildwestshow.com" });
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            context.Employees.AddObject(new Employee {
                   Name = new Name { FirstName = "Bill",
                                     LastName = "Jordan" },
                   Email = "BJordan@wildwestshow.com" });
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            foreach (var employee in
                     context.Employees.OrderBy(e => e.Name.LastName))
            {
                Console.WriteLine("{0}, {1} email: {2}",
                                  employee.Name.LastName,
                                  employee.Name.FirstName,
                                  employee.Email);
            }
        }
    }
}

public class Employee
{
    public int EmployeeId { get; set; }
    public string Email { get; set; }
    public Name Name { get; set; }
}

public class Name
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private ObjectSet<Employee> employees;

    public ObjectSet<Employee> Employees
    {
        get { return employees ?? (employees = CreateObjectSet<Employee>()); }
    }
}
```

The following is the output of the code in Listing 8-6:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Jordan, Bill email: BJordan@wildwestshow.com
Oakley, Annie email: aoakley@wildwestshow.com
```

How It Works

When you use complex types with POCO, keep in mind the following two rules:

- The complex type must be a class. It cannot be a struct.
- Inheritance cannot be used with complex type classes.

In the current version of Entity Framework, complex types do not leverage change tracking. Changes to complex types will not be reflected in change tracking. This means that if you mark the properties on a complex type as **virtual**, there is no change-tracking proxy support. All change tracking is snapshot-based.

When you delete or update a POCO entity with a complex type without first loading it from the database, you need to be careful to create an instance of the complex type. In Entity Framework, instances of complex types are structurally part of the entity, and null values are not supported. The code in Listing 8-7 illustrates one way to handle deletes.

Listing 8-7. Deleting a POCO entity with a complex type

```
int id = 0;
using (var context = new EFRecipesEntities())
{
    var emp = context.Employees.Where(e =>
                e.Name.FirstName.StartsWith("Bill")).FirstOrDefault();
    id = emp.EmployeeId;
}

using (var context = new EFRecipesEntities())
{
    var empDelete = new Employee { EmployeeId = id,
                          Name = new Name { FirstName = string.Empty,
                                            LastName = string.Empty } };
    context.Employees.Attach(empDelete);
    context.Employees.DeleteObject(empDelete);
    context.SaveChanges();
}
```

In Listing 8-7, we first have to find the EmployeeId of Bill Jordan. Because we are trying to show how we would delete Bill without first loading the entity into the context, we create a new context to illustrate deleting Bill given just his EmployeeId. We need to create an instance of the Employee entity complete with the Name type. Because we are deleting, it doesn't matter much what values we put in for FirstName and LastName. The key is that the Name property is not null. We satisfy this requirement by assigning a new (dummy) instance of Name. We then **Attach()** the entity and call **DeleteObject()** and **SaveChanges()**. This deletes the entity.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

8-5. Notifying Entity Framework About Object Changes

Problem

You are using POCO and want to have Entity Framework and the object state manager notified of changes to your objects.

Solution

Let's say you have a model like the one in Figure 8-6.

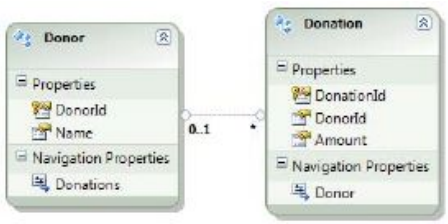


Figure 8-6. *A model for donors and their donations*

This model represents donations and donors. Because some donations are anonymous, the relationship between donor and donation is 0..1 to *.

We want to make changes to our entities, such as moving a donation from one donor to another, and have Entity Framework and the object state manager notified of these changes. In addition, we want Entity Framework to leverage this notification to fix up any relationships that are affected by changes. In our case, if we change the Donor on a Donation, we want Entity Framework to fix up both sides of the relationship. The code in Listing 8-8 demonstrates how to do this.

The key part of Listing 8-8 is that we marked each property as **`virtual`** and each collection returning a type of **`ICollection<T>`**. This allows Entity Framework to create proxies for our POCO entities that enable change tracking.

Listing 8-8. By marking each property as ***`virtual`*** *and each collection a type of* ***`ICollection<T>`****, we get proxies that enable change tracking.*

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            var donation = context.CreateObject<Donation>();
            donation.Amount = 5000M;

            var donor1 = context.CreateObject<Donor>();
            donor1.Name = "Jill Rosenberg";
            var donor2 = context.CreateObject<Donor>();
            donor2.Name = "Robert Hewitt";

            // give Jill the credit for the donation and save
            donor1.Donations.Add(donation);
            context.Donors.AddObject(donor1);
            context.Donors.AddObject(donor2);
            context.SaveChanges();

            // now give Robert the credit
            donation.Donor = donor2;

            // report
            foreach (var donor in context.Donors)
            {
                Console.WriteLine("{0} has given {1} donation(s)", donor.Name,
                                  donor.Donations.Count().ToString());
            }
            var entry = context.ObjectStateManager.GetObjectStateEntry(donation);
            Console.WriteLine("Original Donor Id: {0}",
                              entry.OriginalValues["DonorId"]);
            Console.WriteLine("Current Donor Id: {0}",
                              entry.CurrentValues["DonorId"]);
        }
    }
}

public class Donor
{
    public virtual int DonorId { get; set; }
    public virtual string Name { get; set; }
    public virtual ICollection<Donation> Donations { get; set; }
}

public class Donation
{
    public virtual int DonationId { get; set; }
    public virtual int? DonorId { get; set; }
    public virtual decimal Amount { get; set; }
    public virtual Donor Donor { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    }

    private ObjectSet<Donor> donors;
    private ObjectSet<Donation> donations;

    public ObjectSet<Donor> Donors
    {
        get { return donors ?? (donors = CreateObjectSet<Donor>()); }
    }

    public ObjectSet<Donation> Donations
    {
        get { return donations ?? (donations = CreateObjectSet<Donation>()); }
    }
}
```

The following is the output of the code in Listing 8-8:

```
Jill Rosenberg has given 0 donation(s)

Robert Hewitt has given 1 donation(s)

Original Donor Id: 1

Current Donor Id: 2
```

How It Works

By default, the Entity Framework uses a snapshot-based approach for detecting changes made to POCO entities. If you make some minor code changes to your POCO entities, Entity Framework can create change-tracking proxies that keep the object state manager synchronized with the runtime changes in your POCO entities.

There are two important benefits that come with change-tracking proxies. First, the object state manager stays informed of the changes and can keep the entity object graph state information synchronized with your POCO entities. This means that no time needs to be spent detecting changes using the snapshot-based approach.

Additionally, when the object state manager is notified of changes on one side of a relationship, it can mirror the change on the other side of the relationship if needed. In Listing 8-8, when we moved a Donation from one Donor to another, Entity Framework also fixed up the Donations collections of both Donors.

For the Entity Framework to create the change-tracking proxies for your POCO classes, the following conditions must be met.

- The class must be public, non-abstract, and non-sealed.
- The class must implement virtual getters and setters for all properties that are persisted.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

- You must declare collection-based relationships navigation properties as `ICollection<T>`. They cannot be a concrete implementation or another interface that derives from `ICollection<T>`.

Once your POCO classes have met these requirements, Entity Framework will return instances of the proxies for your POCO classes. If you need to create instances, as we have in Listing 8-8, you need to use the `CreateObject<T>()` method on the object context. This method creates the instance of the proxy for your POCO entity and initializes all the collections as instances of EntityCollection. It is this initialization of your POCO class' collections as instances of EntityCollection that enables fixing up relationships.

8-6. Retrieving the Original (POCO) Object

Problem

You are using POCO and you want to retrieve the original object from a database.

Solution

Let's say you are using a model like the one in Figure 8-7 and you working in a disconnected scenario. You want to use `GetObjectByKey()` to retrieve the original object from the database before you apply changes received from a client. `GetObjectByKey()` takes an entity key. To form an entity key, you'll need the entity set name, which is not available on POCO classes.

Figure 8-7. *A model with a single Item entity*

To get the entity set name, form the entity key, and use `GetObjectByKey()` to retrieve the object and then apply changes, follow the pattern in Listing 8-9.

Listing 8-9. Getting the entity set name to create the entity key enables us to use `GetObjectByKey()`

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        {
            int itemId = 0;
            using (var context = new EFRecipesEntities())
            {
                var item = new Item { Name = "Xcel Camping Tent",
                                      UnitPrice = 99.95M };
                context.Items.AddObject(item);
                context.SaveChanges();

                // keep the item id for the next step
                itemId = item.ItemId;
                Console.WriteLine("Item: {0}, UnitPrice: {1}",
                         item.Name, item.UnitPrice.ToString("C"));
            }

            using (var context = new EFRecipesEntities())
            {
                // pretend this is the updated
                // item we received with the new price
                var item = new Item { ItemId = itemId,
                                Name = "Xcel Camping Tent",
                                UnitPrice = 129.95M };

                // use our method to get the entity set name
                var itemES = context.GetEntitySet(item);

                // create the entity key
                var key = context.CreateEntityKey(itemES.Name, item);

                // retrieve and update the item
                context.GetObjectByKey(key);
                context.ApplyCurrentValues(itemES.Name, item);
                context.SaveChanges();
            }

            using (var context = new EFRecipesEntities())
            {
                var item = context.Items.Single();
                Console.WriteLine("Item: {0}, UnitPrice: {1}", item.Name,
                                  item.UnitPrice.ToString("C"));
            }
        }
    }

    public class Item
    {
        public int ItemId { get; set; }
        public string Name { get; set; }
        public decimal UnitPrice { get; set; }
    }

    public class EFRecipesEntities : ObjectContext
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private ObjectSet<Item> items;
    public ObjectSet<Item> Items
    {
        get { return items ?? (items = CreateObjectSet<Item>()); }
    }

    // gets the entity set
    public EntitySetBase GetEntitySet(Object entityType)
    {
        var container = this.MetadataWorkspace.GetEntityContainer(
                            this.DefaultContainerName,DataSpace.CSpace);
        var entitySet = container.BaseEntitySets.Single(
                es => es.ElementType.Name == entityType.GetType().Name);
        return entitySet;
    }
}
```

The following is the output of the code in Listing 8-9:

```
Item: Xcel Camping Tent, UnitPrice: $99.95
Item: Xcel Camping Tent, UnitPrice: $129.95
```

How It Works

In Listing 8-9, we inserted an item into the model and saved it to the database. Then we pretended to receive an updated item, perhaps from a Silverlight client.

Next, we need to update the item in the database. To do this, we need to get the entity from the database into the context. To get the entity, we'll use **GetObjectByKey()**. To use **GetObjectByKey()**, we need to form the entity key for the entity. Unfortunately, to use **CreateEntityKey()** we need the entity set name. This name is not directly available when we are using POCO classes. To solve this problem, we implemented the **GetEntitySet()** method in our implementation of the object context. This method gets the entity set for our entity. Given the entity set, we can now pass the entity set name to **CreateEntityKey()** to form the entity key. Now armed with the entity key, we can call **GetObjectByKey()** to load the original item and apply the changes with the **ApplyCurrentValues()** method.

If you are using Multiple Entity Sets per Type (MEST), which maps an entity to multiple entity sets, this approach doesn't work. In this recipe, we're assuming that the entity is mapped to just one entity set.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

8-7. Manually Synchronizing the Object Graph and the Object State Manager

Problem

You want to manually control the synchronization between your POCO classes and the object state manager.

Solution

Suppose we have a model for speakers and the talks prepared for various conferences. The model might something like the one in Figure 8-8.

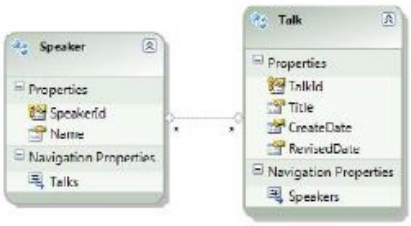


Figure 8-8. *A model with a many-to-many association between speakers and the talks they prepare*

The first thing to note in our model is that Speaker and Talk are in a many-to-many association. We have, through an independent association (and in intermediate SpeakerTalk table in the database), a model that supports many speakers for any given talk and many talks for any given speaker.

We want to manually control the synchronization between our object graph and the object state manager. We will do this by calling the **`DetectChanges()`** method and handling the **`SavingChanges`** event. Along the way, we'll illustrate how the synchronization is progressing.

Follow the pattern in Listing 8-10 to manually synchronize your POCO object graph with the object state manager.

Listing 8-10. Using ***`DetectChanges()`*** *and handling the* ***`SavingChanges`*** *event to manually synchronize the object state manager*

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        {
            var speaker1 = new Speaker { Name = "Karen Stanfield" };
            var talk1 = new Talk { Title = "Simulated Annealing in C#" };
            speaker1.Talks = new List<Talk> { talk1 };

            // associations not yet complete
            Console.WriteLine("talk1.Speaker is null: {0}",
                              talk1.Speakers == null);
            context.Speakers.AddObject(speaker1);

            // now it's fixed up
            Console.WriteLine("talk1.Speaker is null: {0}",
                              talk1.Speakers == null);
            Console.WriteLine("Number of added entries tracked: {0}",
                              context.ObjectStateManager
                              .GetObjectStateEntries(
                                System.Data.EntityState.Added).Count());
            context.SaveChanges();

            // change the talk's title
            talk1.Title = "AI with C# in 3 Easy Steps";
            Console.WriteLine("talk1's state is: {0}",
                              context.ObjectStateManager
                              .GetObjectStateEntry(talk1).State);
            context.DetectChanges();
            Console.WriteLine("talk1's state is: {0}",
                              context.ObjectStateManager
                              .GetObjectStateEntry(talk1).State);

            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            foreach(var speaker in context.Speakers.Include("Talks"))
            {
                Console.WriteLine("Speaker: {0}",speaker.Name);
                foreach(var talk in speaker.Talks)
                {
                    Console.WriteLine("\tTalk Title: {0}",talk.Title);
                }
            }
        }
    }
}

public class Speaker
{
    public int SpeakerId { get; set; }
    public string Name { get; set; }
    public IList<Talk> Talks { get; set; }
}
```


Licensed by Matt Allen 2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
public class Talk
{
    public int TalkId { get; set; }
    public string Title { get; set; }
    public DateTime CreateDate { get; set; }
    public DateTime RevisedDate { get; set; }
    public IList<Speaker> Speakers { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
        this.SavingChanges += new EventHandler(EFRecipesEntities_SavingChanges);
    }

    private ObjectSet<Speaker> speakers;
    private ObjectSet<Talk> talks;

    public ObjectSet<Speaker> Speakers
    {
        get { return speakers ?? (speakers = CreateObjectSet<Speaker>()); }
    }

    public ObjectSet<Talk> Talks
    {
        get { return talks ?? (talks = CreateObjectSet<Talk>()); }
    }

    private void EFRecipesEntities_SavingChanges(object sender, EventArgs e)
    {
        var addedTalks = this.ObjectStateManager
                         .GetObjectStateEntries(
                            System.Data.EntityState.Added)
                         .Where(en => en.Entity is Talk)
                         .Select(en => en.Entity as Talk);
        foreach (var talk in addedTalks)
        {
            talk.CreateDate = DateTime.Now;
            talk.RevisedDate = DateTime.Now;
        }

        var revisedTalks = this.ObjectStateManager
                           .GetObjectStateEntries(
                              System.Data.EntityState.Modified)
                           .Where(en => en.Entity is Talk)
                           .Select(en => en.Entity as Talk);
        foreach (var talk in revisedTalks)
        {
            talk.RevisedDate = DateTime.Now;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }
    }

    public override int SaveChanges(SaveOptions options)
    {
        this.DetectChanges();
        return base.SaveChanges(options);
    }
}
```

The following is the output of the code in Listing 8-10:

```
talk1.Speaker is null: True

talk1.Speaker is null: False

Number of added entries tracked: 3

talk1's state is: Unchanged

talk1's state is: Modified

Speaker: Karen Stanfield

        Talk Title: AI with C# in 3 Easy Steps
```

How It Works

The code in Listing 8-10 is a little involved, so let's take it one step at a time. First off, we create a speaker and a talk. Then we add the talk to the speaker's collection. At this point, the talk is part of the speaker's collection, but the speaker is not part of the talk's collection. The other side of the association has not been fixed up yet.

Next, we add the speaker to the object context with **AddObject(speaker1)**. The second line of the output shows now that the talk's speaker collection is correct. Entity Framework has fixed up the other side of the association. Here Entity Framework did two things. It notified the object state manager that there are three entries to be created (third line of the output). One of these entries is for the speaker, one is for the talk, and one is for the many-to-many association entry. The second thing Entity Framework did was to fix up the talk's speaker collection.

When we call **SaveChanges()**,Entity Framework raises the **SavingChanges** event. Inside the handler for this event we update the CreateDate and RevisedDate properties. After the **SavingChanges** event is handled, Entity Framework calls **DetectChanges()** to find any changes that occurred in the event handler. In Listing 8-10, we override the **SaveChanges()** method and explicitly call **DetectChanges()**, although this is not strictly necessary.

The **DetectChanges()** method relies on a snapshot base comparison of the original and current values for each property on each entity. This process determines what has changed in the object graph. For large object graphs, this comparison process may be time consuming.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

8-8. Testing Domain Objects

Problem

You want to create unit tests for the business rules you have defined for your entities.

Solution

For this solution, we'll use the POCO template to generate the classes for our entities. Using the POCO template will reduce the amount of code we need to write and make the solution a little more clear. Of course, you use the remaining steps in this solution with your hand-crafted POCO classes.

Suppose you have a model like the one shown in Figure 8-9.

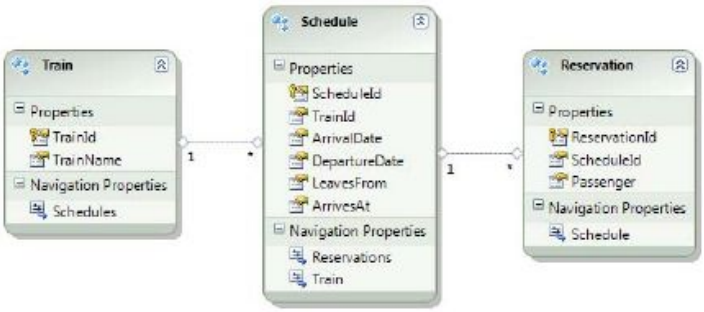


Figure 8-9. *A model of reservations, schedules, and trains*

This model represents reservations for train travel. Each reservation is for a particular scheduled train departure. To create the model and prepare the application for unit testing, do the following:

1. Create an empty solution. Right-click the solution in the Solution Explorer and select Add ➤ New Project. Add a new Class Library project. Name this new project **TrainReservation**.
2. Right-click the TrainReservation project and select Add ➤ New Item. Add a new ADO.NET Entity Data Model. Import the Train, Schedule, and Reservation tables. The resulting model should look like the one in Figure 8-9.
3. Right-click the design surface and select Add Code Generation Item. Select ADO.NET POCO Entity Generator. Name this item **Recipe8.tt**. Click Add. The T4 template for POCO code generation will be added to your project. This will cause POCO classes to be generated for your entities. If the POCO template is not available, use the Tools ➤ Extension Manager to download the template or enable it if you have already installed it.
4. Add the IValidate interface and ChangeAction `enum` in Listing 8-11 to the project.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5. Add the code in Listing 8-12 to the project. This code adds the validation code (the implementation of IValidate) to the Reservation and Schedule classes.

6. Override the **SaveChanges()** method in the object context with the code in Listing 8-13. This will allow us to validate the changes before they are saved to the database.

7. Create the IReservationContext interface in Listing 8-14. We'll use this interface to help us test against a fake object context so that changes are not saved to the real database.

8. The POCO template generates both the POCO classes and the class that implements the object context. We'll need this object context class to implement the IReservationContext interface. To do this, edit the Recipe8.Context.tt template file and add **IReservationContext** at the end of the line that generates the name of the object context class. The complete line should look like the following:

```
<#=Accessibility.ForType(container)#> partial class <#=code.Escape(container)#> :
 ObjectContext,IReservationContext
```

9. The IReservationContext interface returns **IObjectSet<T>**. We need to change the T4 template so that it generates an object context that returns **IObjectSet<T>** in place of **ObjectSet<T>**. To do this, change this line

```
<#=Accessibility.ForReadOnlyProperty(entitySet)#>
 ObjectSet<<#=code.Escape(entitySet.ElementType)#>> <#=code.Escape(entitySet)#>
```

to this

```
<#=Accessibility.ForReadOnlyProperty(entitySet)#>
 IObjectSet<<#=code.Escape(entitySet.ElementType)#>> <#=code.Escape(entitySet)#>
```

and change this line

```
private ObjectSet<<#=code.Escape(entitySet.ElementType)#>>
 <#=code.FieldName(entitySet)#>;
```

to this

```
private IObjectSet<<#=code.Escape(entitySet.ElementType)#>>
 <#=code.FieldName(entitySet)#>;
```

10. Create the repository class in Listing 8-15. This class takes an IReservationContext in the constructor.

11. Right-click the solution and select Add ➤ New Project. Add a Test Project to the solution. Name this new project **Tests**. Add a reference to System.Data.Entity.

12. Create a fake object set and fake object context so that we can test our business rules in isolation without interacting with the database. Use the code in Listing 8-16.

13. We don't want to test against our real database, so we need to create a fake object context that simulates the object context with in-memory collections acting as our data store. Add the unit test code in Listing 8-17 to the Tests project.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The Test project now has three unit tests that exercise the following business rules:

- A passenger cannot have more than one reservation for a scheduled departure.
- The arrival date and time for a schedule must be after the departure date and time.
- The departure location cannot be the same as the arrival location.

Listing 8-11. The IValidate interface

```
public enum ChangeAction
{
    Insert,
    Update,
    Delete
}

interface IValidate
{
    void Validate(ChangeAction action);
}
```

Listing 8-12. Implementation of the IValidate interface for the Reservation and Schedule classes

```
public partial class Reservation : IValidate
{
    public void Validate(ChangeAction action)
    {
        if (action == ChangeAction.Insert)
        {
            if (Schedule.Reservations.Count(r =>
                          r.ReservationId != ReservationId &&
                          r.Passenger == this.Passenger) > 0)
                throw new InvalidOperationException(
                          "Reservation for the passenger already exists");
        }
    }
}

public partial class Schedule : IValidate
{
    public void Validate(ChangeAction action)
    {
        if (action == ChangeAction.Insert)
        {
            if (ArrivalDate < DepartureDate)
            {
                throw new InvalidOperationException(
                          "Arrival date cannot be before departure date");
            }

            if (LeavesFrom == ArrivesAt)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            {
                throw new InvalidOperationException(
                          "Can't leave from and arrive at the same location");
            }
        }
    }
}
```

Listing 8-13. Overriding the ***SaveChanges()*** *method*

```
public partial class EFRecipesEntities
{
    public override int SaveChanges(SaveOptions options)
    {
        this.DetectChanges();
        var entries = from e in this.ObjectStateManager
                       .GetObjectStateEntries(EntityState.Added |
                                              EntityState.Modified |
                                              EntityState.Deleted)
                      where (e.IsRelationship == false) && (e.Entity != null) &&
                            (e.Entity is IValidate)
                      select e;
        foreach (var entry in entries)
        {
            switch (entry.State)
            {
                case EntityState.Added:
                    ((IValidate)entry.Entity).Validate(ChangeAction.Insert);
                    break;
                case EntityState.Modified:
                    ((IValidate)entry.Entity).Validate(ChangeAction.Update);
                    break;
                case EntityState.Deleted:
                    ((IValidate)entry.Entity).Validate(ChangeAction.Delete);
                    break;
            }
        }
        return base.SaveChanges(options & ~SaveOptions.DetectChangesBeforeSave);
    }
}
```

Listing 8-14. We'll use this IReservationContext to define which methods we'll need from the object context

```
public interface IReservationContext : IDisposable
{
    IObjectSet<Train> Trains { get; }
    IObjectSet<Schedule> Schedules { get; }
    IObjectSet<Reservation> Reservations { get; }

    int SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 8-15. The ReservationRepository class that takes an IReservationContext in the constructor

```
public class ReservationRepository
{
    private IReservationContext _context;

    public ReservationRepository(IReservationContext context)
    {
        if (context == null)
            throw new ArgumentNullException("context is null");
        _context = context;
    }
    public void AddTrain(Train train)
    {
        _context.Trains.AddObject(train);
    }

    public void AddSchedule(Schedule schedule)
    {
        _context.Schedules.AddObject(schedule);
    }

    public void AddReservation(Reservation reservation)
    {
        _context.Reservations.AddObject(reservation);
    }

    public void SaveChanegs()
    {
        _context.SaveChanges();
    }

    public List<Schedule> GetActiveSchedulesForTrain(int trainId)
    {
        var schedules = from r in _context.Schedules
                        where r.ArrivalDate.Date >= DateTime.Today &&
                              r.TrainId == trainId
                        select r;
        return schedules.ToList();
    }
}
```

Listing 8-16. The implementation of the fake object set and fake object context

```
public class FakeObjectSet<T> : IObjectSet<T> where T : class
{
    HashSet<T> _data;
    IQueryable _query;

    public FakeObjectSet()
        : this(new List<T>())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
{
}

public FakeObjectSet(IEnumerable<T> initialData)
{
    _data = new HashSet<T>(initialData);
    _query = _data.AsQueryable();
}

public void Add(T item)
{
    _data.Add(item);
}

public void AddObject(T item)
{
    _data.Add(item);
}

public void Remove(T item)
{
    _data.Remove(item);
}

public void DeleteObject(T item)
{
    _data.Remove(item);
}

public void Attach(T item)
{
    _data.Add(item);
}

public void Detach(T item)
{
    _data.Remove(item);
}

Type IQueryable.ElementType
{
    get { return _query.ElementType; }
}

System.Linq.Expressions.Expression IQueryable.Expression
{
    get { return _query.Expression; }
}

IQueryProvider IQueryable.Provider
{
    get { return _query.Provider; }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }

        System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
        {
            return _data.GetEnumerator();
        }

        IEnumerator<T> IEnumerable<T>.GetEnumerator()
        {
            return _data.GetEnumerator();
        }
    }

    public class FakeReservationContext : IReservationContext, IDisposable
    {
        private IObjectSet<Train> trains;
        private IObjectSet<Schedule> schedules;
        private IObjectSet<Reservation> reservations;
        public FakeReservationContext()
        {
            trains = new FakeObjectSet<Train>();
            schedules = new FakeObjectSet<Schedule>();
            reservations = new FakeObjectSet<Reservation>();
        }

        public IObjectSet<Train> Trains
        {
            get { return trains; }
        }

        public IObjectSet<Schedule> Schedules
        {
            get { return schedules; }
        }

        public IObjectSet<Reservation> Reservations
        {
            get { return reservations; }
        }

        public int SaveChanges()
        {
            foreach (var schedule in Schedules.Cast<IValidate>())
            {
                schedule.Validate(ChangeAction.Insert);
            }
            foreach (var reservation in Reservations.Cast<IValidate>())
            {
                reservation.Validate(ChangeAction.Insert);
            }
            return 1;
        }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    public void Dispose()
    {
    }
}
```

Listing 8-17. The unit tests for our Tests project

```
[TestClass]
public class ReservationTest
{
    private IReservationContext _context;

    [TestInitialize]
    public void TestSetup()
    {
        var train = new Train { TrainId = 1, TrainName = "Polar Express" };
        var schedule = new Schedule { ScheduleId = 1, Train = train,
                                      ArrivalDate = DateTime.Now,
                                      DepartureDate = DateTime.Today,
                                      LeavesFrom = "Dallas",
                                      ArrivesAt = "New York" };
        var reservation = new Reservation { ReservationId = 1,
                                            Passenger = "Phil Marlowe",
                                            Schedule = schedule };
        _context = new FakeReservationContext();
        var repository = new ReservationRepository(_context);
        repository.AddTrain(train);
        repository.AddSchedule(schedule);
        repository.AddReservation(reservation);
        repository.SaveChanges();
    }

    [TestMethod]
    [ExpectedException(typeof(InvalidOperationException))]
    public void TestForDuplicateReservation()
    {
        var repository = new ReservationRepository(_context);
        var schedule = repository.GetActiveSchedulesForTrain(1).First();
        var reservation = new Reservation { ReservationId = 2,
                                            Schedule = schedule,
                                            Passenger = "Phil Marlowe" };
        repository.AddReservation(reservation);
        repository.SaveChanges();
    }

    [TestMethod]
    [ExpectedException(typeof(InvalidOperationException))]
    public void TestForArrivalDateGreaterThanDepartureDate()
    {
        var repository = new ReservationRepository(_context);
        var schedule = new Schedule { ScheduleId = 2, TrainId = 1,
                                      ArrivalDate = DateTime.Today,
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                    DepartureDate = DateTime.Now,
                                    ArrivesAt = "New York",
                                    LeavesFrom = "Chicago" };
        repository.AddSchedule(schedule);
        repository.SaveChanges();
    }

    [TestMethod]
    [ExpectedException(typeof(InvalidOperationException))]
    public void TestForArrivesAndLeavesFromSameLocation()
    {
        var repository = new ReservationRepository(_context);
        var schedule = new Schedule { ScheduleId = 3, TrainId = 1,
                                    ArrivalDate = DateTime.Now,
                                    DepartureDate = DateTime.Today,
                                    ArrivesAt = "Dallas",
                                    LeavesFrom = "Dallas" };
        repository.AddSchedule(schedule);
        repository.SaveChanges();
    }
}
```

How It Works

With quite a lot of code, we've manage to build a complete solution that includes an interface (IReservationContext) we can use to abstractly reference an object context, a fake object set (**FakeObjectSet<T>**), a fake object context (FakeReservationContext), and a small set of unit tests. We use the fake object context so that our tests don't interact with the database. The purpose of the tests is to validate our business rules, not to test the database interactions.

One key to the solution is that we created a simplified repository that managed inserting and selecting our objects. The constructor for this repository takes an IReservationContext. This subtle abstraction allows us to pass in an instance of any class that implements IReservationContext. To test our domain objects, we pass in an instance of FakeReservationContext. To allow our domain objects to be persisted to the database, we would pass in an instance of our real object context: EFRecipesEntities.

We need the object sets returned by our fake object context to match the object sets returned by the real EFRecipesEntities object context. To do this, we changed the T4 template that generates the context to return **IObjectSet<T>** in place of **ObjectSet<T>**. We made sure our fake object context also returned object sets of type **IObjectSet<T>**. With this in place, we implemented our **FakeObjectSet<T>** and derived it from **IObjectSet<T>**.

In the Tests project, we set up the tests by creating a ReservationRepository based on an instance of the FakeReservationContext. The unit tests interact with the FakeReservationContext in place of the real object context.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Best Practice

There are two testing approaches that seem to work well for Entity Framework:

1. Define a repository interface that both the real repository and one or more "testing" repositories implement. By hiding all the interactions with the persistence framework behind the implementation of the repository interface, there is no need to create fake versions of any of the other infrastructure parts. This can simplify the implementation of the testing code, but it may leave parts of the repository itself untested.
2. Define an interface for the object context that exposes properties of type `IObjectSet<T>` and a `SaveChanges()` method, as we have done in this recipe. The real object context and all the fake object contexts must implement this interface. Using this approach, you don't need to fake the entire repository, which may be difficult in some cases. Your fake object contexts don't need to mimic the behavior of the entire ObjectContext class; that would be a real challenge. You do need to limit your code to just what is available on the interfaces.

8-9. Testing a Repository Against a Database

Problem

You want to test your repository against the database.

Solution

You have created a repository that manages all the queries, inserts, updates, and deletes. You want to test this repository against a real instance of the underlying database. Suppose you have a model like the one shown in Figure 8-10. Because we will create and drop the database during the tests, let's start from the beginning in a test database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

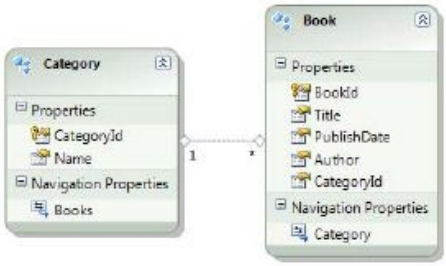


Figure 8-10. *A model of books in categories*

To test your repository, do the following:

1. Create an empty solution. Right-click the solution in the Solution Explorer and select Add ➤ New Project. Add a new Class Library project. Name this new project **BookRepository**.
2. Create a new database. Call the database **Test**. We'll create and drop this database in the unit tests, so make sure you create a new empty database.
3. Add the Book and Category tables along with the relation corresponding to the model in Figure 8-10. Import these tables into a new model. Alternatively, you can use Model First to create the model; then generate the database script to create the database.
4. Add the code in Listing 8-18. This will create a BookRepository class that handles inserts and queries against the model.
5. Right-click the solution and select Add ➤ New Project. Select Test Project from the installed templates. Add a reference to System.Data.Entity and a project reference to BookRepository.
6. Right-click the Test project and select Add ➤ New Test. Add a Unit Test to the Test project. Add the code in Listing 8-19 to create the tests.
7. Right-click the Test project and select Add ➤ New Item. Select Application Configuration File from the General templates. Copy the <connectionStrings> section from the App.config file in the BookRepository project and insert it into the new App.config file in the Test project.
8. Right-click the Test project and select Set as Startup Project. Select Debug ➤ Start Debugging or press F5 to execute the tests. Make sure there are no active connections to the Test database. Active connections will cause the `DropDatabase()` method to fail.

Listing 8-18. The BookRepository class that handles inserts and queries against the model

```
namespace BookRepository
{
    public class BookRepository
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        private TestEntities _context;

        public BookRepository(TestEntities context)
        {
            _context = context;
        }

        public void InsertBook(Book book)
        {
            _context.Books.AddObject(book);
        }

        public void InsertCategory(Category category)
        {
            _context.Categories.AddObject(category);
        }

        public void SaveChanges()
        {
            _context.SaveChanges();
        }

        public IQueryable<Book> BooksByCategory(string name)
        {
            return _context.Books.Where(b => b.Category.Name == name);
        }

        public IQueryable<Book> BooksByYear(int year)
        {
            return _context.Books.Where(b => b.PublishDate.Year == year);
        }
    }
}
```

Listing 8-19. BookRepositoryTest class with the unit tests

```
[TestClass]
public class BookRepositoryTest
{
    private TestEntities _context;

    [TestInitialize]
    public void TestSetup()
    {
        _context = new TestEntities();
        if (_context.DatabaseExists())
        {
            _context.DeleteDatabase();
        }
        _context.CreateDatabase();
    }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    [TestMethod]
    public void TestsBooksInCategory()
    {
        var repository = new BookRepository.BookRepository(_context);
        var construction = new Category { Name = "Construction" };
        var book = new Book { Title = "Building with Masonary",
                              Author = "Dick Kreh",
                              PublishDate = new DateTime(1998, 1, 1) };
        book.Category = construction;
        repository.InsertCategory(construction);
        repository.InsertBook(book);
        repository.SaveChanges();

        // test
        var books = repository.BooksByCategory("Construction");
        Assert.AreEqual(books.Count(), 1);
    }

    [TestMethod]
    public void TestBooksPublishedInTheYear()
    {
        var repository = new BookRepository.BookRepository(_context);
        var construction = new Category { Name = "Construction" };
        var book = new Book { Title = "Building with Masonary",
                              Author = "Dick Kreh",
                              PublishDate = new DateTime(1998, 1, 1) };
        book.Category = construction;
        repository.InsertCategory(construction);
        repository.InsertBook(book);
        repository.SaveChanges();

        // test
        var books = repository.BooksByYear(1998);
        Assert.AreEqual(books.Count(), 1);
    }
}
```

How It Works

There are two common approaches to testing that are used with Entity Framework. The first approach is to test the business logic implemented in your objects. For this approach, we test against a "fake" database layer because our focus is on the business logic that governs the interactions of the objects and the rules that apply just before objects are persisted to the database. We illustrated this approach in Recipe 8-8.

A second approach is to test both the business logic and the persistence layer by interacting with the real database. This approach is more extensive, but also more costly in term of time and resources. When implemented in an automated test harness, like the ones we often use in a continuous integration environment, we need to automate the creation and dropping of the test database. Each test iteration should start with a database in a known clean state. Subsequent test runs should not be affected by

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

residue left in the database by previous tests. Dropping and creating databases together with the end-to-end code exercise requires more resources than the business logic only testing illustrated in Recipe 8-8.

In the unit tests in Listing 8-19 we checked to see whether the database exists in the Test Initialize phase. If the database exists, it is dropped with the **DropDatabase()** method. Next, we create the database with the **CreateDatabase()** method. These methods use the connection string contained in the App.config file. This connection string would likely be different from the development database connection string. For simplicity, we used the same connection string for both.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 9

Using the Entity Framework in N-Tier Applications

Not all applications can be neatly bundled into a single process. In fact, in this ever increasingly networked world, many application architectures have both the classic logical layers of presentation, application, and data, but also are physically deployed across multiple computers. While logical layering can be accommodated in a single Application Domain without much concern for proxies, marshalling, serialization, and network protocols, applications that span from something small like a Windows Phone 7 Series device to servers found in your data center, need to take all these into account. Fortunately, the Entity Framework together with technologies like Microsoft's Windows Communication Foundation is well suited for these n-tier applications.

In this chapter, we'll cover a wide range of recipes for using the Entity Framework in n-tier applications. In these recipes, we'll cover the basic create, read, update, and delete operations you'll use in virtually all your n-tier applications. Additionally, we'll take a deep dive into the use of self-tracking entities, entity and proxy serialization, concurrency, and working with the unique challenges of tracking entity changes outside the scope of an object context.

9-1. Deleting an Entity When Disconnected

Problem

You have an object that you have retrieved from a Windows Communication Foundation (WCF) service and you want to mark it for deletion.

Solution

Suppose you have a model like the one in Figure 9-1.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

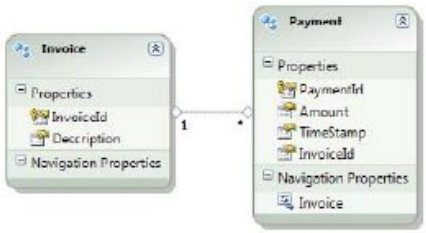


Figure 9-1. *A model for payments on invoices*

Our model represents payments on invoices. In our application, we have implemented a WCF service to handle the database interactions from a client. We want to delete an object, in our case a Payment entity, using the service. To keep the solution as simple as possible, we'll build a WCF service library and define the model inside of it by doing the following.

1. Create a WCF Service Library by right-clicking the solution and selecting Add New Project. Select WCF ➤ WCF Service Library.
2. Right-click the project and select Add New Item. Select Data ➤ ADO.NET Entity Data Model. Use the wizard to add a model with the Invoice and Payment tables. For simplicity, we've removed the Payments navigation property on the Invoice entity. Right-click the TimeStamp property in the Payment entity, select Properties and set its Concurrency Mode to Fixed.
3. Because the default entities generated by Entity Framework can't be serialized, we'll need to create our own entities using POCO. With the Entity Data Model Designer open, view the properties window and change the Code Generation Strategy to None. Next, add the code in Listing 9-1 to create the POCO classes.
4. In the IService1.cs file, change the service definition as shown in Listing 9-2.
5. In the Service1.cs file, implement the service as shown in Listing 9-3.
6. To test our service, we'll need a client. Add a new Windows Console Application project to the solution. Use the code in Listing 9-4 for the client. Add a service reference to the client by right-clicking the client project and selecting Add Service Reference. You may need to right-click the service project and select Debug ➤ Start Instance to start an instance of your service before you can add a service reference in the client.

Listing 9-1. POCO classes that can be serialized by our WCF service

```
public class Payment
{
    public int PaymentId { get; set; }
    public decimal Amount { get; set; }
    public byte[] TimeStamp { get; set; }
    public int InvoiceId { get; set; }
    public Invoice Invoice { get; set; }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
public class Invoice
{
    public int InvoiceId { get; set; }
    public string Description { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private ObjectSet<Payment> payments;
    private ObjectSet<Invoice> invoices;

    public ObjectSet<Payment> Payments
    {
        get { return payments ?? (payments = CreateObjectSet<Payment>()); }
    }

    public ObjectSet<Invoice> Invoices
    {
        get { return invoices ?? (invoices = CreateObjectSet<Invoice>()); }
    }
}
```

Licensed by Matt Allen 2175280

Listing 9-2. The service contract for our WCF service

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    Payment InsertPayment();

    [OperationContract]
    void DeletePayment(Payment payment);
}
```

Listing 9-3. The implementation of our service contract

```
public class Service1 : IService1
{
    public Payment InsertPayment()
    {
        using (var context = new EFRecipesEntities())
        {
            // delete the previous test data
            context.ExecuteStoreCommand("delete from chapter9.payment");
            context.ExecuteStoreCommand("delete from chapter9.invoice");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            var payment = new Payment { Amount = 99.95M, Invoice =
                                new Invoice { Description = "Auto Repair" } };
            context.Payments.AddObject(payment);
            context.SaveChanges();
            return payment;
        }
    }

    public void DeletePayment(Payment payment)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Payments.Attach(payment);
            context.Payments.DeleteObject(payment);
            context.SaveChanges();
        }
    }
}
```

Listing 9-4. A simple console application to test our WCF service

```
class Program
{
    static void Main(string[] args)
    {
        var client = new Service1Client();
        var payment = client.InsertPayment();
        client.DeletePayment(payment);
    }
}
```

If you set a breakpoint on the first line in the **Main()** method of the client and debug the application, you can step through the insertion and deletion of a Payment entity.

How It Works

In the client, we use the **InsertPayment()** method to insert a new payment into the database. The method returns the payment that was inserted. The payment that is returned is disconnected from the object context. In fact, the object context may be in a different process space or on an entirely different computer.

We use the **DeletePayment()** method to delete the payment entity from the database. In the implementation of this method (Listing 9-3), we first **Attach()** the payment entity. In the object context, this entity is now in an unchanged state. Calling the **DeleteObject()** method marks the object for deletion. **SaveChanges()** deletes the payment from the database.

The payment object that we attached had all its properties set as they were when the object was inserted into the database. However, because we're using foreign key association, only the entity key and the concurrency property, in our case the TimeStamp property, are needed for Entity Framework to generate the appropriate **where** clause to delete the entity. The one exception to this rule is when your POCO class has one or more properties that are complex types. Because complex types are considered

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

structural parts of an entity, they cannot be null. For these, you would simply create a dummy instance of the complex type. If you leave the complex type property null, `SaveChanges()` will throw an exception.

If you are using an independent association in which the multiplicity of the related entity is one or 0..1, then Entity Framework requires the entity keys of those references to be set correctly in order to generate the appropriate `where` clause of an update or delete statement. In our example, if we had an independent association between Invoice and Payment, we would need to set the Invoice navigation property to an instance of Invoice with the correct value for the InvoiceId property. The resulting `where` clause would include the PaymentId, the TimeStamp, and the InvoiceId.

If your object has several independent associations, setting all of them usually becomes tedious. You might find it simpler to retrieve the instance from the database and then mark it for deletion. This makes your code a little simpler, but when you retrieve the object from the database, Entity Framework will rewrite the query to bring in all the relationships that are one or 0..1 unless, of course, you are using the NoTracking context option. In our case, when you load the payment entity prior to marking for deletion, Entity Framework will create an object state entry for the payment entity and a relationship entry for the relationship between Payment and Invoice. When we mark the payment entity for deletion, Entity Framework will also mark the relationship entry for deletion. As before, the resulting `where` clause would include the PaymentId, the TimeStamp, and the InvoiceId.

Another option for deleting entities in independent associations is to eagerly load the related entities and transport the entire object graph back to the WCF service for deletion. In our case, we could eagerly load the related invoice entity with the payment entity. When we delete the payment entity, we could send back the graph containing both entities to the service. Of course, this approach consumes more network bandwidth and processing time for serialization, but it might make the code somewhat more clear.

9-2. Managing Concurrency When Disconnected

Problem

You want to make sure that changes made on an entity by a WCF client are applied only if the concurrency token has not changed.

Solution

Let's suppose you have a model like the one in Figure 9-2.

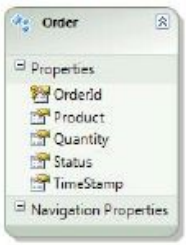


Figure 9-2. *Our model with a single Order entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

We want to update an order using a WCF service while guaranteeing that the order we're updating has not changed since the last time we retrieved the order. We'll show two slightly different ways to handle this. In both approaches we use a concurrency column, in our case, the TimeStamp column.

1. Create a WCF Service Library by right-clicking the solution and selecting Add New Project. Select WCF ➤ WCF Service Library.
2. Right-click the project and select Add New Item. Select Data ➤ ADO.NET Entity Data Model. Use the wizard to add a model with the Order table. Right-click the TimeStamp property, select Properties and set its Concurrency Mode to Fixed.
3. Because the default entities generated by Entity Framework can't be serialized, we'll need to create our own entities using POCO. With the Entity Data Model Designer open, view the properties window and change the Code Generation Strategy to None. Next, add the code in Listing 9-5 to create the POCO class for the Order entity and our object context.
4. In the IService1.cs file, change the service definition as shown in Listing 9-6.
5. In the Service1.cs file, implement the service as shown in Listing 9-7.
6. To test our service, we'll need a client. Add a new Windows Console Application project to the solution. Use the code in Listing 9-8 for the client. Add a service reference to the client by right-clicking the client project and selecting Add Service Reference. You may need to right-click the service project and select Debug ➤ Start Instance to start an instance of your service before you can add a service reference in the client.

Listing 9-5. The code for the Order POCO class and the related object context

```
public class Order
{
    public int OrderId { get; set; }
    public string Product { get; set; }
    public int Quantity { get; set; }
    public string Status { get; set; }
    public byte[] TimeStamp { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private IObjectSet<Order> orders;

    public IObjectSet<Order> Orders
    {
        get { return orders ?? (orders = CreateObjectSet<Order>()); }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 9-6. Our WCF service contract

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    Order InsertOrder();

    [OperationContract]
    void UpdateOrderWithoutRetrieving(Order order);

    [OperationContract]
    void UpdateOrderByRetrieving(Order order);
}
```

Listing 9-7. The implementation of our service contract

```
public class Service1 : IService1
{
    public Order InsertOrder()
    {
        using (var context = new EFRecipesEntities())
        {
            // remove previous test data
            context.ExecuteStoreCommand("delete from chapter9.[order]");

            var order = new Order { Product = "Camping Tent",
                                    Quantity = 3, Status = "Received" };
            context.Orders.AddObject(order);
            context.SaveChanges();
            return order;
        }
    }

    public void UpdateOrderWithoutRetrieving(Order order)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Orders.Attach(order);
            if (order.Status == "Received")
            {
                var entry = context.ObjectStateManager
                                .GetObjectStateEntry(order);
                entry.SetModifiedProperty("Quantity");
                context.SaveChanges();
            }
        }
    }

    public void UpdateOrderByRetrieving(Order order)
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        using (var context = new EFRecipesEntities())
        {
            var dbOrder = context.Orders
                            .Single(o => o.OrderId == order.OrderId);
            if (dbOrder != null &&
                StructuralComparisons.StructuralEqualityComparer
                      .Equals(order.TimeStamp, dbOrder.TimeStamp))
            {
                dbOrder.Quantity = order.Quantity;
                context.SaveChanges();
            }
        }
    }
}
```

Listing 9-8. The client we use to test our WCF service

```
class Program
{
    static void Main(string[] args)
    {
        var service = new Service1Client();
        var order = service.InsertOrder();
        order.Quantity = 5;
        service.UpdateOrderWithoutRetrieving(order);
        order = service.InsertOrder();
        order.Quantity = 3;
        service.UpdateOrderByRetrieving(order);
    }
}
```

If you set a breakpoint on the first line in the **Main()** method of the client and debug the application, you can step through the inserting the order and updating the order using both methods.

How It Works

Our **InsertOrder()** method (Listing 9-7) deletes any previous test data, inserts a new order, and returns the order. The order returned has both the database generated OrderId and TimeStamp properties. In our client, we use two slightly different approaches to update this order.

In the first approach, **UpdateOrderWithoutRetrieving()**, we **Attach()** the order from the client, check whether the order status is "Received" and if it is, we mark the entity's Quantity property as modified, and call **SaveChanges()**.Entity Framework will generate an update statement setting the new quantity with a **where** clause that includes both the OrderId and the TimeStamp values from the order entity. If the TimeStamp value has changed by some intermediate update to the database, this update will fail. This ensures that the order entity we are updating has not been modified between the time we obtained it from the **InsertOrder()** method and the time we updated in the database.

Although the first approach is fairly simple and efficient, there are times that you may need to know before you call **SaveChanges()** that the entity has been changed. In these cases, you can retrieve the entity from the database and manually compare the TimeStamp properties to determine whether an intervening change has occurred. This is illustrated with a fresh order by the **UpdateOrderByRetrieving()**

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

method. Although not foolproof (the order could be changed by another client between the time you retrieve the order from the database and compare TimeStamps, and call **SaveChanges()**), this approach does provide valuable insight into what properties or associations have changed on an entity. This may be particularly useful if the object graph or entities are large or complex.

9-3. Finding Out What Has Changed

Problem

You have the original version of an object and a modified version. You want to update the object in the model using Entity Framework to determine which properties have changed.

Solution

Let's suppose you have a model like the one in Figure 9-3.

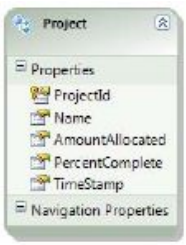


Figure 9-3. *A model with an Project entity*

Given an original order entity and a modified order entity, you want to update the model to reflect the changes in the modified order entity. We'll put together a simple web page that updates the database. Do the following:

1. Create a new empty Web Application project. Add an ADO.NET Entity Data Model with the Order table to the web application project. The model should look like the one in Figure 9-3. Make sure you set the TimeStamp property's Concurrency Mode to Fixed. You can do this on the properties page for TimeStamp.
2. We'll use a simplified Repository pattern with an ObjectDataSource control for the web page. Add the code in Listing 9-9 to create the repository.
3. We'll create just the details part of a typical master/detail page. Add an ASP.NET web page with the code in Listing 9-10.
4. Add the code behind in Listing 9-11. This code will populate the model with some test data.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 9-9. A project repository with the ***UpdateProject()*** *method*

```
public class ProjectRepository
{
    public ProjectRepository()
    {
    }

    public List<Project> RetrieveAll()
    {
        using (var context = new EFRecipesEntities())
        {
            return context.Projects.ToList();
        }
    }

    public void UpdateProject(Project project, Project originalProject)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Projects.Attach(project);
            context.Projects.ApplyOriginalValues(originalProject);
            context.SaveChanges();
        }
    }
}
```

Listing 9-10. A DetailsView in an ASP.NET web page

```
<body>
  <form id="form1" runat="server">
    <div>
      <h2>Azle City Project</h2>
      <asp:DetailsView ID="DetailsView1" runat="server"
          DataKeyNames="ProjectId, TimeStamp"
          AutoGenerateRows="false" DataSourceID="projectSource">
        <Fields>
          <asp:BoundField DataField="ProjectId" HeaderText="ProjectId"
             ReadOnly="true" />
          <asp:BoundField DataField="Name" HeaderText="Name" ReadOnly="true" />
          <asp:BoundField DataField="AmountAllocated"
             HeaderText="Amount Allocated" />
          <asp:BoundField DataField="PercentComplete"
             HeaderText="Percent Complete" />
          <asp:CommandField ShowEditButton="true" />
        </Fields>
      </asp:DetailsView>

      <asp:ObjectDataSource ID="projectSource" runat="server"
        UpdateMethod="UpdateProject"
        SelectMethod="RetrieveAll" TypeName="Recipe3.ProjectRepository"
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        ConflictDetection="CompareAllValues"
        DataObjectTypeName="Recipe3.Project"
        OldValuesParameterFormatString="original{0}" />
    </div>
  </form>
</body>
```

Listing 9-11. The code behind for the ASP.NET page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            using (var context = new EFRecipesEntities())
            {
                // delete any previous test data
                context.ExecuteStoreCommand("delete from chapter9.project");

                // insert some test data
                context.Projects.AddObject(new Project {
                    Name = "Trim City Park Trees",
                    AmountAllocated = 8200M,
                    PercentComplete = 75 });
                context.SaveChanges();
            }
        }
    }
}
```

Figure 9-4 shows the web page as rendered in a browser. The Edit button from the DetailsView control allows editing of the properties which is shown in Figure 9-5.

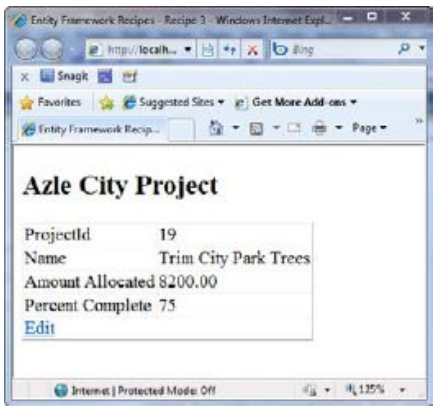


Figure 9-4. *The DetailsView for a project*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

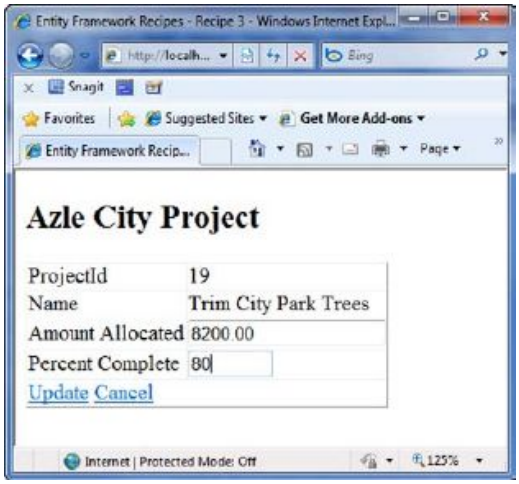


Figure 9-5. *Updating the PercentComplete property to 80%*

How It Works

In the DetailsView control we set the DataKeyNames to ProjectId and TimeStamp. These values are serialized in the control state and preserved on post back. On update, the ASP.NET page will create a new project instance and assign the ProjectId and TimeStamp values from the control state. The remaining properties will be populated from the control. These represent the possibly changed values.

On the ObjectDataSource control we set the ConflictDetection to CompareAllValues. This causes ASP.NET to store the original values of the project entity in the viewstate. During update, ASP.NET will send the modified project entity and the project entity with the original values to the update method. The OldValuesParameterFormatString attribute is used to determine the name of the method parameter to receive the original project entity. In our case, the parameter's name is originalProject. We set the format string to original{0} which will format the correct name.

In the `UpdateProject()` method in Listing 9-9 we show how to update the project entity with the new values given both the original project instance and the updated instance. We `Attach()` the modified instance to the object context and then call the `ApplyOriginalValues()` method. This method marks all the properties in the attached instance as modified if they differ from the corresponding properties on the instance passed in to the method. With the changed properties marked as modified, we call `SaveChanges()` to send the update to the database. This approach works well even if our modified entity has several related entities. The `Attach()` method will attach the object graph.

Another approach is to attach the original instance and then call the `ApplyCurrentValues()` method. In some sense, this is the mirror image of the first approach. The `ApplyCurrentValues()` method updates the instance in the context with the property values that are different in the updated instance. If you have an object graph associated with the original instance, this approach may be more useful.

Both approaches only compare changes in the scalar properties of the entities. If you are using foreign key associations, then the one or 0..1 navigation properties will reflect the changes. Navigation properties that are collections will not be compared.

In many cases, especially in disconnected scenarios, after changes are made to entities, we receive only the updated entity. This would happen in our example if we had not set the ConflictDetection

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

attribute on the ObjectDataSource control to CompareAllValues. To perform an update in these cases, we first need to load the original entity before applying the changes. This approach is covered in Recipe 2 in this chapter.

9-4. Using POCO With WCF

Problem

You want to use POCO with WCF for selects, inserts, deletes, and updates.

Solution

Licensed by Matt Allen 2175280

Let's say you have a model like the one in Figure 9-6.

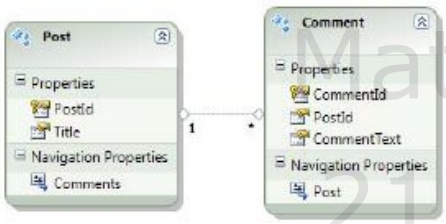


Figure 9-6. *A model for blog posts and comments*

Our model represents blog posts and the comments that readers have about the posts. To make things clearer, we've stripped out most of the properties we would normally have such as the body of the post, the author, the date and time of the post, and so on.

We want to put the all the database code behind a WCF service so that clients can read, update, and delete posts and comments as well as insert new ones. To create the service, do the following.

1. Add a class library project to your solution. Add an ADO.NET Entity Data Model with the Post and Comment tables. Because we are going to use POCO, right-click the .edmx file and view the Properties. With the Entity Data Model Designer open, view the properties window and change the Code Generation Strategy to None.
2. Add the code in Listing 9-12 to create the POCO classes and the object context. You may choose to put these in a separate project, add them to the class library project, or split them into two new projects: one for the Post and Comment classes and one for the EFReceipesEntities class.
3. Add a WCF Service Application to the solution. Use the default name Service1 just to keep things simple.
4. Change the IService1.cs file to reflect the new IService1 interface in Listing 9-13.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5. Change the service application code in the Service1.svc.cs file using the code in Listing 9-14. Add a project reference to the class library and a **using** statement so that the references to the POCO classes resolve correctly. You will also need to add a reference to System.Data.Entity as well as a using for System.Data.Objects.
6. Copy the connection string from the App.Config file in the class library to the web.config in the service application.
7. Add a Windows Console Application to the project. We'll use this for our client to test the WCF service. Use the code in Listing 9-15 for the client. Right-click the console application project and select Add Service Reference and add a reference to the Service1 service. You will also need to add a project reference to the class library created in Step 1.

Listing 9-12. Our POCO classes Post, Comment, and our EFReceipesEntities object context

```
[DataContract(IsReference = true)]
public class Post
{
    [DataMember]
    public virtual int PostId { get; set; }
    [DataMember]
    public virtual string Title { get; set; }
    [DataMember]
    public virtual ICollection<Comment> Comments { get; set; }
}

[DataContract(IsReference=true)]
public class Comment
{
    [DataMember]
    public virtual int CommentId { get; set; }
    [DataMember]
    public virtual int PostId { get; set; }
    [DataMember]
    public virtual string CommentText { get; set; }
    [DataMember]
    public virtual Post Post { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    private ObjectSet<Post> posts;
    private ObjectSet<Comment> comments;

    public ObjectSet<Post> Posts
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    {
        get { return posts ?? (posts = CreateObjectSet<Post>()); }
    }

    public ObjectSet<Comment> Comments
    {
        get { return comments ?? (comments = CreateObjectSet<Comment>()); }
    }
}
```

Listing 9-13. The service contract for our service

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    void Cleanup();

    [OperationContract]
    Post GetPostByTitle(string title);

    [OperationContract]
    Post SubmitPost(Post post);

    [OperationContract]
    Comment SubmitComment(Comment comment);

    [OperationContract]
    void DeleteComment(Comment comment);
}
```

Listing 9-14. The implementation of the service contract in Listing 9-13 (be sure to add references to System.Data.Entity and System.Security to this project)

```
public class Service1 : IService
{
    public void Cleanup()
    {
        using (var context = new EFRecipesEntities())
        {
            context.ExecuteStoreCommand("delete from chapter9.comment");
            context.ExecuteStoreCommand("delete from chapter9.post");
        }
    }

    public Post GetPostByTitle(string title)
    {
        using (var context = new EFRecipesEntities())
        {
            context.ContextOptions.ProxyCreationEnabled = false;
            var post = context.Posts.Include("Comments")
                              .Single(p => p.Title == title);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            return post;
        }
    }

    public Post SubmitPost(Post post)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Posts.Attach(post);
            if (post.PostId == 0)
            {
                // this must be an insert
                context.ObjectStateManager
                       .ChangeObjectState(post, EntityState.Added);
            }
            else
            {
                context.ObjectStateManager
                       .ChangeObjectState(post, EntityState.Modified);
            }
            context.SaveChanges();
            return post;
        }
    }

    public Comment SubmitComment(Comment comment)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Comments.Attach(comment);
            if (comment.CommentId == 0)
            {
                // this is an insert
                context.ObjectStateManager
                       .ChangeObjectState(comment, EntityState.Added);
            }
            else
            {
                var entry = context.ObjectStateManager
                                   .GetObjectStateEntry(comment);
                entry.SetModifiedProperty("CommentText");
            }
            context.SaveChanges();
            return comment;
        }
    }

    public void DeleteComment(Comment comment)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Comments.Attach(comment);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            context.Comments.DeleteObject(comment);
            context.SaveChanges();
        }
    }
}
```

Listing 9-15. Our Windows console application that serves as our test client

```
class Program
{
    static void Main(string[] args)
    {
        using (var client = new ServiceReference1.Service1Client())
        {
            // cleanup previous data
            client.Cleanup();

            // insert a post
            var post = new Post { Title = "POCO Proxies" };
            post = client.SubmitPost(post);

            // update the post
            post.Title = "Change Tracking Proxies";
            client.SubmitPost(post);

            // add a comment
            var comment1 = new Comment {
                 CommentText = "Virtual Properties are cool!",
                 PostId = post.PostId };
            var comment2 = new Comment {
                 CommentText = "I use ICollection<T> all the time",
                 PostId = post.PostId };
            comment1 = client.SubmitComment(comment1);
            comment2 = client.SubmitComment(comment2);

            // update a comment
            comment1.CommentText = "How do I use ICollection<T>?";
            client.SubmitComment(comment1);

            // delete comment 1
            client.DeleteComment(comment1);

            // get posts with comments
            var p = client.GetPostByTitle("Change Tracking Proxies");
            Console.WriteLine("Comments for post: {0}", p.Title);
            foreach (var comment in p.Comments)
            {
                Console.WriteLine("\tComment: {0}", comment.CommentText);
            }
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following is the output of our test client in Listing 9-15:

```
Comments for post: Change Tracking Proxies
        Comment: I use ICollection<T> all the time
```

How It Works

Let's start with the Windows console application, which is our test client for the service. We create an instance of our service client in a **using {}** block. Just as we've done with the creating an instance of an object context in a **using {}** block, this ensures that **Dispose()** is called when we leave the block either normally or via an exception.

Once we have an instance of our service client, the first thing we do is call the **Cleanup()** method. We do this to remove any previous test data we might have.

With the next couple of lines, we call the service's **SubmitPost()** method. In this method's implementation (see Listing 9-14), we **Attach()** the post and then check whether the PostId is 0. If the PostId is 0, then we assume it's a new post. Admittedly, this is a rather crude way of determining whether the post is new. It depends on the domain of the valid Ids for a post as well as the runtime initializing integers to 0. A better approach might involve sending an additional parameter to the method or creating a separate **InsertPost()** method. The best approach is dependent on the structure of your application.

If the post is to be inserted, we change the object state of the post to **EntityState.Added**. Otherwise, we change its object state to **EntityState.Modified**. These mark the object as new; an insert statement should be generated, or if modified, an update statement should be generated. If the post is inserted, the post instance's PostId is updated with the new correct value. The post is returned.

Inserting and updating a comment is similar to inserting and updating a post with one significant difference. As a business rule, when we update a comment, we want to make sure to only update the CommentText property. This property holds the body of the comment and we don't want to update any other part of the comment. To do this, we mark just the CommentText property as modified. Entity Framework will generate an update statement that changes just the CommentText column in the database.

To delete a comment, we **Attach()** the comment and call **DeleteObject()**, which marks the comment for deletion.

Finally, the **GetPostByTitle()** method eagerly loads the comments for each post. This returns the object graph of posts and related comments. Notice that we turn off proxy creation with the line **ProxyCreationEnabled = false**. This is important because we don't want change tracking proxies created for our entities. These would normally be created because we have marked the entities' properties as virtual. These proxies cannot be serialized and if they were generated, we would get an error message similar to the following:

```
The underlying connection was closed: The connection was closed unexpectedly
```

In this recipe, we've seen that we can use POCO to handle CRUD operations with WCF. Because there is no state information stored on the client, we've built separate methods for inserting/updating and deleting posts and comments. In the next recipe, we'll look at using Self-Tracking entities with WCF. This approach reduces the number of methods our service must implement and simplifies the communication between the client and the server.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

9-5. Using Self-Tracking Entities With WCF

Problem

You want to use self-tracking entities with a WCF service to perform inserts, updates, and deletes.

Solution

Let's suppose you have a model like the one in Figure 9-7.

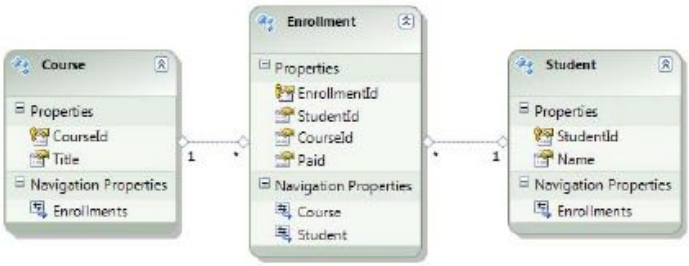


Figure 9-7. *A model for students and the courses in which they are enrolled*

Our model represents students and the courses in which they are enrolled. We want to create a WCF service that handles the interaction with the model and the database using self-tracking entities. To do this, we need to use the Self-Tracking Entities template. We also want to isolate the entity classes in a separate project so that both the client and the service can reference entities project. This is useful when using Self-Tracking Entities with WCF because if we simply use the service reference on the client side to get access to the entity classes, we will not be using entities generated by the Self-Tracking Entities template. Much of the complexity in the following steps involves creating four separate projects: EnrollmentData for the model and context, EnrollmentEntities for the generated entity classes, EnrollmentService for the WCF service, and EnrollmentClient for the test client.

To create our application, do the following.

1. Add a Class Library project to your solution and name it EnrollmentData. Add an ADO.NET Entity Data Model with the Course, Enrollment, and Student tables.
2. With the Entity Data Model Designer open, view the properties window and change the Code Generation Strategy to None.
3. Right-click the design surface again and select Add Code Generation Item. Select Code under the Installed Templates and then select the ADO.NET Self-Tracking Entity Generator template. Name the new template **Enrollment.tt**. This will add two templates to the project: Enrollment.tt and Enrollment.Context.tt. The first template generates the entities and the second template generates the object context.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Add a Class Library project to the solution. Call this new project **EnrollmentEntities**. Move the Enrollment.tt template from the EnrollmentData project to the EnrollmentEntities project. Add a reference to System.Runtime.Serialization. Add a project reference in the EnrollmentData project to the EnrollmentEntities project.
5. Because we've moved the Enrollment.tt template file we need to edit it to change the reference to the .edmx file for the model. Edit the Enrollment.tt template and change the line `string inputFile = @"Recipe5.edmx"` to `string inputFile = @"..\EnrollmentData\Recipe5.edmx"`. You may have named your .edmx file something else, if so, make the changes so that the relative path is correct to your .edmx file.
6. Edit the Enrollment.Context.tt template (which should still be in the EnrollmentData project) and add using `EnrollmentEntities`; after each <auto-generated> comment section. This will put the using statement in each generated file.
7. Add a WCF Service Application project to the solution. Name the new service **EnrollmentService**. Add a reference to System.Data.Entity. Add project references to EnrollmentEntities and EnrollmentData. Copy the <connectionStrings> section from the App.Config file in EnrollmentData to the Web.config.
8. Change the IService1.cs file to reflect the new IService1 interface in Listing 9-16.
9. Change the IService1.svc.cs file to reflect the new implementation of the IService1 interface in Listing 9-17.
10. Add a Windows Console Application to the solution. Name the project **EnrollmentClient**. Use the code in Listing 9-18 for this application. Add a project reference to EnrollmentEntities and a service reference to the EnrollmentService.

Listing 9-16. The new IService1 interface, which replaces the code in the IService1.cs file

```
using EnrollmentEntities;
namespace EnrollmentService
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        void InsertTestRecord();

        [OperationContract]
        Student SubmitStudentEnrollment(Student student);

        [OperationContract]
        List<Course> GetCourseDetail();
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 9-17. The implementation of the IService1 interface, which replaces the code in the IService.svc.cs file

```
using EnrollmentData;
using EnrollmentEntities;
namespace EnrollmentService
{
    public class Service1 : IService1
    {
        public void InsertTestRecord()
        {
            using (var context = new EFRecipesEntities())
            {
                // remove previous test data
                context.ExecuteStoreCommand("delete from chapter9.enrollment");
                context.ExecuteStoreCommand("delete from chapter9.course");
                context.ExecuteStoreCommand("delete from chapter9.student");

                // insert new test data
                var student = new Student { Name = "Robin Rosen",
                                            StudentId = 1 };
                var course1 = new Course { Title = "Mathematical Logic 101",
                                           CourseId = 1 };
                var course2 = new Course { Title = "Organic Chemistry 211",
                                           CourseId = 2 };
                context.Students.AddObject(student);
                context.Courses.AddObject(course1);
                context.Courses.AddObject(course2);
                context.SaveChanges();
            }
        }

        public Student SubmitStudentEnrollment(Student student)
        {
            using (var context = new EFRecipesEntities())
            {
                context.Students.ApplyChanges(student);
                context.SaveChanges();
                student.AcceptChanges();
                foreach (var enrollment in student.Enrollments)
                {
                    enrollment.AcceptChanges();
                }
                return student;
            }
        }

        public List<Course> GetCourseDetail()
        {
            using (var context = new EFRecipesEntities())
            {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                return context.Courses.Include("Enrollments.Student").ToList();
            }
        }
    }
}
```

Listing 9-18. The EnrollmentClient test client code

```
using EnrollmentEntities;
using EnrollmentClient.ServiceReference1;
namespace EnrollmentClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var client = new Service1Client())
            {
                // insert test data
                client.InsertTestRecord();

                // create some entities with known
                // a known student id and course id
                var student1 = new Student { StudentId = 1 };
                var course1 = new Course { CourseId = 1 };
                student1.MarkAsUnchanged();
                course1.MarkAsUnchanged();

                // enroll the student in two courses, using both
                // an entity and the foreign key
                student1.Enrollments.Add(new Enrollment { Course = course1,
                                                          Paid = true });
                student1.Enrollments.Add(new Enrollment { CourseId = 2,
                                                          Paid = false });

                // save the enrollments
                student1 = client.SubmitStudentEnrollment(student1);

                // now drop courses the student has not paid for
                foreach (var enrollment in student1.Enrollments
                                       .Where(e => !e.Paid).ToArray())
                {
                    enrollment.MarkAsDeleted();
                }

                // student got married!, change name
                student1.Name += "Robin Rosen-Parker";
                client.SubmitStudentEnrollment(student1);

                // retrieve the courses and enrollments
                foreach (var course in client.GetCourseDetail())
                {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                    Console.WriteLine("Course: {0}", course.Title);
                    foreach (var en in course.Enrollments)
                    {
                        Console.WriteLine("\tStudent: {0} {1} paid",
                                en.Student.Name, en.Paid ? "has" : "has not");
                    }
                }
            }
        }
    }
}
```

The following is the output of the test client:

```
Course: Mathematical Logic 101

        Student: Robin Rosen-Parker has paid

Course: Organic Chemistry 211
```

Licensed by Matt Allen 2175280

How It Works

In the model, we turned off the default code generation and used the Self-Tracking Entities template to generate both the entity classes and the object context. These self-tracking entities can record changes on scalar and complex properties as well as references, and navigation properties without the help of Entity Framework. This makes self-tracking entities particularly useful in client-side scenarios where an entity is modified outside of the scope of an object context. This is typically the case in Silverlight, WCF, and some ASP.NET applications.

Self-tracking entities track changes because they implement the **IObjectChangeTracker** interface. When the server receives the changed objects, we use the **ApplyChanges()** extension method on the object context to replay the changes. This is clean and quite powerful and frees us from implementing lots of code to determine what properties on which objects have changed. The downside is that these changes are replayed blindly. On a complex object graph, we may have changes that occur throughout the graph. Some validation is likely needed before the **SaveChanges()** method is called. We'll cover validating self-tracking entities in Recipe 6.

A self-tracking entity has a change tracking property that can be used to access the change state of the entity. When you create a self-tracking entity, its initial change state is Added. When you modify the entity, its change state is set to Modified. To change the state manually, use one of the extension methods **MarkAsUnchanged()**, **MarkAsModified()**, **MarkAsDeleted()**, and **MarkAsAdded()**. In Listing 9-18, we used the **MarkAsUnchanged()** method to change the status of the Student and Course entities to the Unchanged state instead of querying from the server. This is similar to calling **Attach()** to attach an existing object to an object context.

By default, self-tracking entities do not have change tracking enabled. You have to set ChangeTrackingEnable to **true** before changes are tracked. When using WCF, the following code generated by the template takes care of enabling change tracking when the entities are deserialized using the DataContract serialization.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
[OnDeserialized]
public void OnDeserializedMethod(StreamingContext context)
{
    IsDeserializing = false;
    ChangeTracker.ChangeTrackingEnabled = true;
}
```

9-6. Validating Self-Tracking Entities

Problem

On the server side, you have received an object graph containing self-tracking entities and you want to validate these changes before `SaveChanges()` is called.

Solution

Let's suppose you have a model like the one in Figure 9-8.

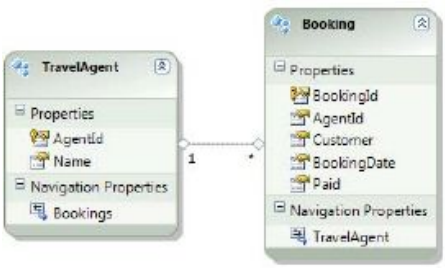


Figure 9-8. *A model for travel bookings and the associated travel agents*

1. Add a Class Library project to your solution and name it **BookingData**. Add an ADO.NET Entity Data Model with the TravelAgent and Booking tables.
2. Right-click the design surface and view the Properties With the Entity Data Model Designer open, view the properties window and change the Code Generation Strategy to None.
3. Right-click the design surface again and select Add Code Generation Item. Select Code under the Installed Templates and then select the ADO.NET Self-Tracking Entity Generator template. Name the new template **Booking.tt**. This will add two templates to the project: Booking.tt and Booking.Context.tt. The first template generates the entities and the second template generates the object context.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Add a Class Library project to the solution. Call this new project **BookingEntities**. Move the Booking.tt template from the BookingData project to the BookingEntities project. Add a reference to System.Runtime.Serialization. Add a project reference in the BookingData project to the BookingEntities project.
5. Because we've moved the Booking.tt template file, we need to edit it to change the reference to the .edmx file for the model. Edit the Booking.tt template and change the line `string inputFile = @"Recipe6.edmx"` to `string inputFile = @"..\BookingData\Recipe6.edmx"`. You may have named your .edmx file something else, if so, make the changes so that the relative path is correct to your .edmx file.
6. Edit the Booking.Context.tt template (which should still be in the BookingData project) and add `using BookingEntities;` after each <auto-generated> comment section. This will put the using statement in each generated file.
7. Add a WCF Service Application project to the solution. Name the new service **BookingService**. Add a reference to System.Data.Entity. Add project references to BookingEntities and BookingData. Copy the <connectionStrings> section from the App.Config file in BookingData to the Web.config file.
8. Change the IService1.cs file to reflect the new IService1 interface in Listing 9-19.
9. Change the IService1.svc.cs file to reflect the new implementation of the IService1 interface in Listing 9-20.
10. Add a Windows Console Application to the solution. Name the project **BookingClient**. Use the code in Listing 9-21 for this application. Add a project reference to BookingEntities and a service reference to the BookingService.

Listing 9-19. The new IService1 interface, which replaces the code in the IService1.cs file

```
using BookingEntities;
namespace BookingService
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        TravelAgent GetAgentWithBookings();

        [OperationContract]
        void SubmitAgentBookings(TravelAgent agent);
    }
}
```

Listing 9-20. The implementation of the IService1 interface, which replaces the code in the IService.svc.cs file

```
using BookingData;
using BookingEntities;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
namespace BookingService
{
    public class Service1 : IService1
    {
        public TravelAgent GetAgentWithBookings()
        {
            InsertAgent();
            using (var context = new EFRecipesEntities())
            {
                return context.TravelAgents.Include("Bookings")
                              .Single(a => a.Name == "John Tate");
            }
        }

        public void SubmitAgentBookings(TravelAgent agent)
        {
            using (var context = new EFRecipesEntities())
            {
                ValidateAgentBeforeApplyChanges(agent);
                context.TravelAgents.ApplyChanges(agent);
                ValidateAgentAfterApplyChanges(context);
                context.SaveChanges();
            }
        }

        private void ValidateAgentAfterApplyChanges(EFRecipesEntities context)
        {
            var cantDelete = context.ObjectStateManager
                      .GetObjectStateEntries(EntityState.Deleted)
                      .Any(e => e.Entity is Booking && ((Booking)e.Entity).Paid);
            ValidateCondition(cantDelete,
                 "Can't delete a booking that is paid for.");

            var cantBook = context.ObjectStateManager
                      .GetObjectStateEntries(EntityState.Added)
                      .Any(e => e.Entity is Booking &&
                        ((Booking)e.Entity).BookingDate
                              .Subtract(DateTime.Today).Days > 20);
            ValidateCondition(cantBook,
                 "Can't book more than 20 days in advance.");
        }

        private void ValidateAgentBeforeApplyChanges(TravelAgent agent)
        {
            var cantAddOrDelete =
              agent.ChangeTracker.State == ObjectState.Deleted ||
              agent.ChangeTracker.State == ObjectState.Deleted;
            ValidateCondition(cantAddOrDelete, "Can't add or delete an agent.");

            var cantModify = agent.Bookings
                   .Any(b => b.ChangeTracker.State == ObjectState.Modified &&
                         b.BookingDate < DateTime.Today);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            ValidateCondition(cantModify, "Can't modify an expired booking.");
        }

        private static void ValidateCondition(bool condition, string message)
        {
            if (condition)
            {
                throw new FaultException<InvalidOperationException>(
                  new InvalidOperationException(message), message);
            }
        }

        private void InsertAgent()
        {
            using (var context = new EFRecipesEntities())
            {
                // delete any previous test data
                context.ExecuteStoreCommand("delete from chapter9.booking");
                context.ExecuteStoreCommand("delete from chapter9.travelagent");

                // inser the test data
                var agent = new TravelAgent { Name = "John Tate" };
                var booking = new Booking {Customer = "Karen Stevens",
                                   Paid = false,
                                   BookingDate = DateTime.Parse("2/2/2010")};
                agent.Bookings.Add(booking);
                context.TravelAgents.AddObject(agent);
                context.SaveChanges();
            }
        }
    }
}
```

Listing 9-21. The BookingClient test client code

```
using BookingClient.ServiceReference1;
using BookingEntities;
namespace BookingClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var client = new Service1Client())
            {
                var agent = client.GetAgentWithBookings();
                agent.Bookings.Add(new Booking {
                           BookingDate = DateTime.Today.AddDays(5),
                           Customer = "Jan Thomas", Paid = true});
                agent.Name = "John Parker";

                var bookings = agent.Bookings
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                .Where(b => b.Customer == "Karen Stevens").ToList();
            foreach (var booking in bookings)
            {
                booking.MarkAsDeleted();
            }
            client.SubmitAgentBookings(agent);
        }
    }
  }
}
```

How It Works

In Listing 9-20, we use two different approaches to validating the self-tracking entity object graph. In the first approach, we validate the object graph before calling **ApplyChanges()**. This means we validate the objects before the changes are applied to the object context. With this approach you can find changes that would violate your business rules before these changes affect the object context. The drawback to this approach is that your service needs intimate knowledge of the object graph, and these validation rules need to be tailored to work with the ChangeTracker.State values. This second requirement limits the validation rules to self-tracking entities.

In the second approach, we validate the objects after the changes have been applied to the object context. Here we are not concerned with the structure of the object graph or the ChangeTracker.State values. We can apply these rules with or without the use of self-tracking entities.

9-7. Using Self-Tracking Entities on the Server Side

Problem

You want to use self-tracking entities with ASP.NET to keep track of changes while the context is not around.

Solution

Let's suppose you have a mode like the one in Figure 9-9.

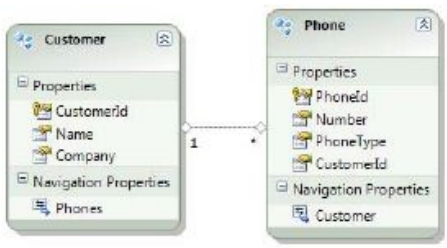


Figure 9-9. *A model for customers and their phone numbers*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

We want to use self-tracking entities with ASP.NET to keep track of changes on entities without an object context. When we need to save the changes, we create a new object context and replay the changes recorded by the self-tracking entities.

Do the following to create a web application that uses self-tracking entities to track changes without an object context:

1. Create a new empty Web Application project. Add an ADO.NET Entity Data Model with the Customer and Phone tables to the web application project. The model should look like the one in Figure 9-9.
2. Right-click the design surface and select Add Code Generation Item. Select Code under the Installed Templates and then select the ADO.NET Self-Tracking Entity Generator template. Name the new template **Recipe7.tt**.
3. Add the CustomerRepository class in Listing 9-22. We'll use the methods in this class to get a customer and to update customers and phone numbers.
4. In the CustomerRepository, we use the **StartSelfTracking()** method on the context. We'll need to implement this method. Add a partial class for our EFRecipesEntities class using the code in Listing 9-23. This code also overrides the **SaveChanges()** method so that we can mark our self-tracking entities as UnChanged after they are saved.
5. To test our code, create an ASP.NET page with code in Listing 9-24 and the code behind in Listing 9-25.

Listing 9-22. The CustomerRepository class we use getting and updating customers

```
public class CustomerRepository : IDisposable
{
    private EFRecipesEntities context;

    public CustomerRepository()
    {
        this.context = new EFRecipesEntities();
    }

    public void Dispose()
    {
        this.context.Dispose();
    }

    public Customer GetCustomer(string name)
    {
        var customer = this.context.Customers
                           .Include("Phones").Single(c => c.Name == name);
        this.context.StartSelfTracking();
        return customer;
    }

    public Customer SubmitCustomerWithPhones(Customer customer)
    {
        this.context.Customers.ApplyChanges(customer);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        this.context.SaveChanges();
        return customer;
    }
}
```

Listing 9-23. We extend the EFRecipesEntities class with this partial class

```
public partial class EFRecipesEntities
{
    public void StartSelfTracking()
    {
        var entities = this.ObjectStateManager
                        .GetObjectStateEntries(~EntityState.Detached)
                        .Where(e => !e.IsRelationship)
                        .Select(e => e.Entity)
                        .OfType<IObjectWithChangeTracker>();
        foreach (var entity in entities)
        {
            entity.StartTracking();
        }
    }

    public override int SaveChanges(SaveOptions options)
    {
        var affected = base.SaveChanges(options);
        if (SaveOptions.AcceptAllChangesAfterSave ==
            (SaveOptions.AcceptAllChangesAfterSave & options))
        {
            var entities = this.ObjectStateManager
                     .GetObjectStateEntries(EntityState.Unchanged)
                     .Where(e => !e.IsRelationship)
                     .Select(e => e.Entity)
                     .OfType<IObjectWithChangeTracker>();
            foreach (var entity in entities)
            {
                entity.AcceptChanges();
            }
        }
        return affected;
    }
}
```

Listing 9-24. Our ASP.NET page demonstrating reading, creating, and updating a customer

```
<body>
  <form id="form1" runat="server">
    <div>
      <asp:Button ID="button1" Text="Create Customer"
                  OnClick="CreateCustomer" runat="server" />
      <br />
      <asp:Button ID="button2" Text="Read Customer"
                  OnClick="ReadCustomer" runat="server" />
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
      <br />
      <asp:Button ID="button3" Text="Update Customer"
                  OnClick="UpdateCustomer" runat="server" />
      <br />
      <h2>Customer Details</h2>
      <table>
        <tr>
          <td>Customer Name</td>
          <td><asp:Label ID="CustomerName" runat="server" /></td>
        </tr>
        <tr>
          <td>Phone Number</td>
          <td><asp:Label ID="PhoneNumber" runat="server" /></td>
        </tr>
      </table>
    </div>
  </form>
</body>
```

Listing 9-25. The code behind for our ASP.NET page

```
public partial class Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            using (var context = new EFRecipesEntities())
            {
                // delete previous test data
                context.ExecuteStoreCommand("delete from chapter9.phone");
                context.ExecuteStoreCommand("delete from chapter9.customer");
            }
        }
    }

    private void ShowCustomer()
    {
        if (this.Session["Customer"] != null)
        {
            Customer customer = (Customer)this.Session["Customer"];
            this.CustomerName.Text = customer.Name;
            if (customer.Phones.Count > 0)
            {
                this.PhoneNumber.Text = customer.Phones[0].Number;
            }
        }
    }
    protected void CreateCustomer(object sender, EventArgs e)
    {
        var customer = new Customer { Name = "Phillip Marlowe",
                                      Company = "Chandler Enterprises" };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

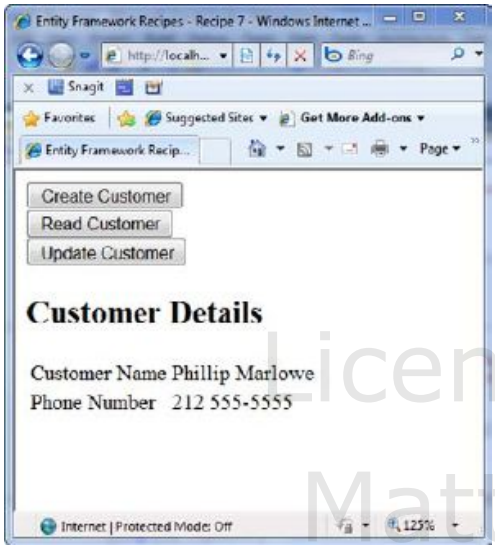
```
        customer.Phones.Add(new Phone { Number = "212 555-5555",
                                        PhoneType = "Office" });
        using (var repository = new CustomerRepository())
        {
            repository.SubmitCustomerWithPhones(customer);
        }
    }

    protected void ReadCustomer(object sender, EventArgs e)
    {
        using (var repository = new CustomerRepository())
        {
            this.Session["Customer"] =
                          repository.GetCustomer("Phillip Marlowe");
        }
        ShowCustomer();
    }

    protected void UpdateCustomer(object sender, EventArgs e)
    {
        Customer customer = (Customer)this.Session["Customer"];
        var number = customer.Phones
                             .FirstOrDefault(p => p.PhoneType == "Office");
        if (number != null)
            number.MarkAsDeleted();
        customer.Phones.Add(new Phone { Number = "817 867-5309",
                                        PhoneType = "Cell" });
        using (var repository = new CustomerRepository())
        {
            var cust = repository.SubmitCustomerWithPhones(customer);
            cust.StartTracking();
            this.Session["Customer"] = cust;
        }
        ShowCustomer();
    }
}
```

To test the code, click the Create Customer button. Nothing interesting is changed in the browser, but this causes our test data to be inserted into the database.

Once the test data is inserted, click the Read Customer button. This reads the customer entity and displays it on the page (see Figure 9-10).

Finally, click the Update Customer button. This changes the customer's phone number and displays the updated customer and phone number (see Figure 9-11).

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



Licensed by Matt Allen 2175280

Figure 9-10. *The web page after the customer is created and the customer is read and stored in the session state*

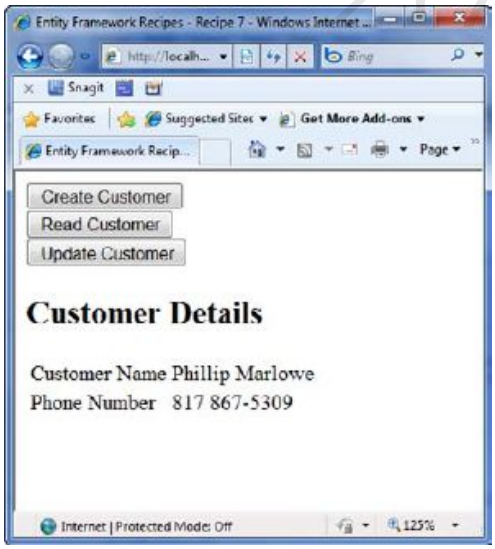


Figure 9-11. *The web page after the customer has been updated and the new updated customer has been inserted into the session state*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Self-tracking entities make n-tier development easier because they implement their own change tracking mechanism. Self-tracking entities record changes without the support of an object context. When sent to the server, these changes can be replayed in the presence of an object context using the **ApplyChanges()** method.

To make this process easier in WCF scenarios, the generated code for self-tracking entities hooks into the deserialization process using the OnDeserializedAttribute. This code turns on change tracking before the entity is received by the client. This makes change tracking with self-tracking entities fairly seamless in a WCF environment.

The story is a little more manual outside of WCF. In this recipe, we used the self-tracking entities in an ASP.NET environment. To turn on change tracking, we implemented the **StartSelfTracking()** method. In this method we find all the self-tracking entities in the object state manager and for each call **StartTracking()**. This sets the ChangeTracker.EnableChangeTracking property to **true** and initiates change tracking for the entity.

Change tracking is also enabled on an entity in two other cases. It is enabled when the entity is attached to another entity that has change tracking enabled. It is also enabled when the state of the self-tracking entity is changed with **MarkAsUnchanged()**, **MarkAsModified()**, or **MarkAsDeleted()**.

Change tracking is disabled or stopped on a self-tracking entity when you call **StopTracking()** or after you apply the changes on the entity to object context using **ApplyChanges()**. The reason why the internal mechanism for change tracking stops in a self-tracking entity when **ApplyChanges()** is called is that at this point the entity is attached to the object context. The object context is now responsible for tracking changes. As with any other POCO entity, after changes have been made you can call **DetectChanges()** to cause snapshot-based change tracking or you can simply call **SaveChanges()** which internally calls **DetectChanges()**.

For other entities, **SaveChanges()** changes the state of each entity to Unchanged. However, for self-tracking entities, **SaveChanges()** leaves the state as Added. In Listing 9-23 we override the **SaveChanges()** method to manually change the state of the self-tracking entities (those that implement **IObjectWithChangeTracker**).

When the Create Customer button is clicked, the code behind (Listing 9-25) creates our test customer entity and calls **SubmitCustomerWithPhones()** to insert the new customer and the phone numbers for the customer. When a new customer entity is created, change tracking is disabled. We don't need to turn it on at this point because the entity is in the Added state and we have no changes to apply.

When the Read Customer button is clicked, the customer is read from the repository using the **GetCustomer()** method. This method turns on change tracking for the entities by calling our **StartSelfTracking()** method (see Listing 9-23).

When the Update Customer button is clicked, the code behind reads the customer from the session state, makes some changes to the customer's phones, and submits these changes using the CustomerRepository's **SubmitCustomerWithPhones()** method. The customer entity that is returned has change tracking disabled. We explicitly enable this with **StartTracking()** before we assign the customer back to the session state.

You might be tempted to use the **ObjectMaterialized** event to turn on tracking for all entities when they are materialized. This is not a viable approach because the **ObjectMaterialized** event only guarantees that regular properties are initialized before the event is raised. Collection properties might change after the event has been raised. If you turn on change tracking while handling the **ObjectMaterialized** event, you may start tracking changes on collections that are simply part of the normal object materialization process.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

9-8. Serializing Proxies in a WCF Service

Problem

You have a dynamic proxy object returned from a query. You want to serialize the proxy as a plain old CLR object.

Solution

Let's suppose you have a model like the one in Figure 9-12.

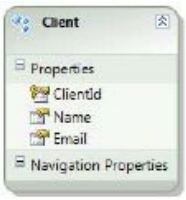


Figure 9-12. *A model with a Client entity*

We'll use the ProxyDataContractResolver class to deserialize a proxy object on the server to a POCO object on the WCF client. Do the following:

1. Create a new WCF Service Application. Add an ADO.NET Entity Data Model with the Client table. The model should look like the one in Figure 9-12.
2. With the Entity Data Model Designer open, view the properties window and change the Code Generation Strategy to None.
3. Create the Client POCO class and object context from the code in Listing 9-26.
4. We need the DataContractSerializer to use ProxyDataContractResolver to resolve the client proxy to the client entity for the WCF service's client. For this, we'll create an operation behavior attribute and apply the attribute on the `GetClient()` service method. Add the code in Listing 9-27 to create the new attribute.
5. Change the IService1.cs interface using the code in Listing 9-28.
6. Change the implementation of the IService1 interface in the IService1.svc.cs file with the code in Listing 9-29.
7. Add a Windows Console Application to the solution. This will be our test client. Use the code in Listing 9-30 to implement our test client. Add a service reference to our WCF service.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 9-26. Our Client POCO class and our object context

```
public class Client
{
    public virtual int ClientId { get; set; }
    public virtual string Name { get; set; }
    public virtual string Email { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    private ObjectSet<Client> clients;
    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
    }

    public ObjectSet<Client> Clients
    {
        get { return clients ?? (clients = CreateObjectSet<Client>()); }
    }
}
```

Listing 9-27. Our custom operation behavior attribute

```
using System.ServiceModel.Description;
using System.ServiceModel.Channels;
using System.ServiceModel.Dispatcher;
using System.Data.Objects;

namespace Recipe8
{
    public class ApplyProxyDataContractResolverAttribute : Attribute,
                IOperationBehavior
    {
        public void AddBindingParameters(OperationDescription description,
                                         BindingParameterCollection parameters)
        {
        }

        public void ApplyClientBehavior(OperationDescription description,
                                        ClientOperation proxy)
        {
            DataContractSerializerOperationBehavior
                dataContractSerializerOperationBehavior =
                      description.Behaviors
                       .Find<DataContractSerializerOperationBehavior>();
            dataContractSerializerOperationBehavior.DataContractResolver =
                      new ProxyDataContractResolver();
        }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        public void ApplyDispatchBehavior(OperationDescription description,
                    DispatchOperation dispatch)
        {
            DataContractSerializerOperationBehavior
                dataContractSerializerOperationBehavior =
                      description.Behaviors
                       .Find<DataContractSerializerOperationBehavior>();
            dataContractSerializerOperationBehavior.DataContractResolver =
                      new ProxyDataContractResolver();
        }

        public void Validate(OperationDescription description)
        {
        }
    }
}
```

Listing 9-28. Our IService1 interface definition, which replaces the code in IService1.cs

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    void InsertTestRecord();

    [OperationContract]
    Client GetClient();

    [OperationContract]
    void Update(Client client);
}
```

Listing 9-29. The implementation of the IService1 interface, which replaces the code in IService1.svc.cs

```
public class Client
{
    [ApplyProxyDataContractResolver]
    public Client GetClient()
    {
        using (var context = new EFRecipesEntities())
        {
            context.ContextOptions.LazyLoadingEnabled = false;
            return context.Clients.Single();
        }
    }

    public void Update(Client client)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Clients.Attach(client);
            context.ObjectStateManager.ChangeObjectState(client,
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                    EntityState.Modified);
            context.SaveChanges();
        }
    }

    public void InsertTestRecord()
    {
        using (var context = new EFRecipesEntities())
        {
            // delete previous test data
            context.ExecuteStoreCommand("delete from chapter9.client");

            // insert new test data
            context.ExecuteStoreCommand(@"insert into
                      chapter9.client(Name, Email)
                      values ('Jerry Jones','jjones@gmail.com')");
        }
    }
}
```

Listing 9-30. Our Windows console application test client

```
using Recipe8Client.ServiceReference1;

namespace Recipe8Client
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var serviceClient = new Service1Client())
            {
                serviceClient.InsertTestRecord();
                var client = serviceClient.GetClient();
                Console.WriteLine("Client is: {0} at {1}",
                                  client.Name, client.Email);
                client.Name = "Alex Park";
                client.Email = "AlexP@hotmail.com";
                serviceClient.Update(client);
                client = serviceClient.GetClient();
                Console.WriteLine("Client changed to: {0} at {1}",
                                  client.Name, client.Email);
            }
        }
    }
}
```

The following is the output of our test client:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Client is: Jerry Jones at jjones@gmail.com
Client changed to: Alex Park at AlexP@hotmail.com
```

How It Works

Microsoft recommends using POCO with WCF so that serialization is easier and lazy loading is not triggered. Self-tracking entities are an ideal solution for WCF applications because they are essentially POCO classes with the added plumbing for managing change tracking. However, if your application is using POCO objects with changed-based notification (you have marked properties as **virtual** and navigation property collections are of type ICollection), then Entity Framework will create dynamic proxies for entities returned from queries.

There are two problems with dynamic proxies and WCF. The first problem has to do with the serialization of the proxy. The DataContractSerializer can only serialize and deserialize known types, such as our Client entity. However, we need to serialize the proxy for the Client, not the Client. Here is where DataContractResolver comes to the rescue. It can map one type to another during serialization. ProxyDataContractResolver derives from DataContractResolver and maps proxy types to POCO classes such as our Client entity. To use the ProxyDataContractResolver, we created an attribute (see Listing 9-27) to resolve proxies into POCO classes. We applied this attribute to the **GetClient()** method in Listing 9-29. This causes the dynamic proxy for the Client entity returned by the **GetClient()** to be correctly serialized for its journey to the user of the WCF service.

The second problem with dynamic proxies and WCF has to do with lazy loading. When the DataContractSerializer serializes the entity, it accesses each of the properties of the entity that would trigger lazy loading of navigation properties. This, of course, is not what we want. To prevent this, we explicitly turned off lazy loading in Listing 9-29.

9-9. Serializing Self-Tracking Entities in the ViewState

Problem

You want to save a self-tracking entity in the ViewState of an ASP.NET application.

Solution

Let's suppose you have a model like the one in Figure 9-13.

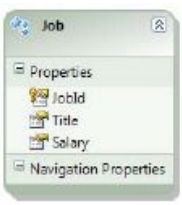


Figure 9-13. *A model with a Job entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

If you try to assign a job entity to the ViewState, the BinaryFormatter would fail. To get around this, we can create our own serializer using the DataContractSerializer to serialize the self-tracking entity. To create your own serializer and a test page, do the following:

1. Create a new empty Web Application. Add an ADO.NET Entity Data Model with the Job table. Right-click the design surface and view the Properties.
2. Right-click the design surface and select Add Code Generation Item. Select the ADO.NET Self-Tracking Entity Generator from the Code Installed Template. Click Add.
3. Add the code in Listing 9-31 to the project. We'll use this class to serialize the object graph.
4. Add a Default.aspx page to the project. Use the code in Listing 9-32 for the page and the code in Listing 9-33 for the code behind.

Listing 9-31. The ByteArraySerializer class is used to serialize the object graph.

```
using System.IO;
using System.Runtime.Serialization;

namespace Recipe9
{
    public class ByteArraySerializer
    {
        public static byte[] ToByteArray<T>(T graph)
        {
            var stream = new MemoryStream();
            var serializer = new DataContractSerializer(typeof(T));
            serializer.WriteObject(stream, graph);
            return stream.ToArray();
        }

        public static T ToObject<T>(byte[] bytes)
        {
            var stream = new MemoryStream(bytes);
            stream.Position = 0;
            var serializer = new DataContractSerializer(typeof(T));
            return (T)serializer.ReadObject(stream);
        }
    }
}
```

Listing 9-32. The ASP.NET page that tests our ViewState serialization

```
<body>
  <form id="form1" runat="server">
    <div>
      <table>
        <tr>
          <td>New Job Title:</td>
          <td><asp:TextBox ID="JobTitle" runat="server" /></td>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        </tr>
        <tr>
          <td>New Salary:</td>
          <td><asp:TextBox ID="Salary" runat="server" /></td>
        </tr>
      </table>
      <br />
      <asp:Button ID="create" runat="server" OnClick="Create_Click"
           Text="Create Job" />  
      <asp:Button ID="update" runat="server" OnClick="Update_Click"
           Text="Update Job" />

      <table>
        <tr>
          <td>Job Title:</td>
          <td><asp:Label ID="JobTitleLabel" runat="server" /></td>
        </tr>
        <tr>
          <td>Salary:</td>
          <td><asp:Label ID="SalaryLabel" runat="server" /></td>
        </tr>
      </table>

    </div>
  </form>
</body>
```

Listing 9-33. The code behind for the page

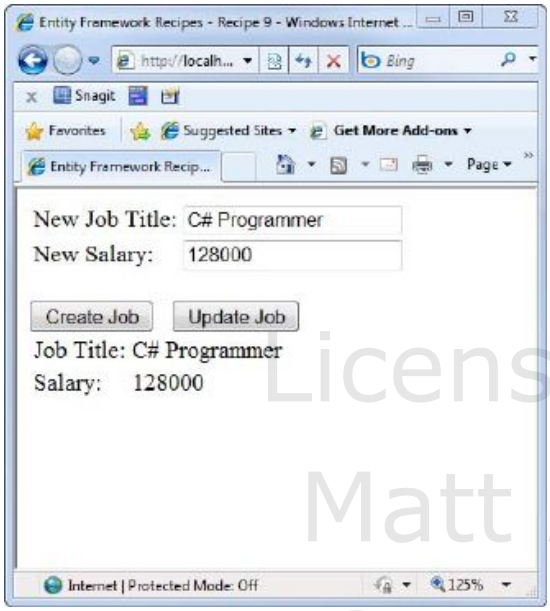
```
public partial class Default : System.Web.UI.Page
{
    public Job Job
    {
        get
        {
            var bytes = ViewState["job"] as byte[];
            return ByteArraySerializer.ToObject<Job>(bytes);
        }

        set
        {
            var bytes = ByteArraySerializer.ToByteArray<Job>(value);
            ViewState["job"] = bytes;
        }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
        if (!this.IsPostBack)
        {
            // create the default job
            this.Job = CreateJob("Plumber", 82000M);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            InitializeControls();
        }

    }

    private void InitializeControls()
    {
        this.JobTitleLabel.Text = Job.Title;
        this.SalaryLabel.Text = Job.Salary.ToString();
    }

    private Job CreateJob(string title, decimal salary)
    {
        using (var context = new EFRecipesEntities())
        {
            return new Job { Title = title, Salary = salary };
        }
    }

    protected void Create_Click(object sender, EventArgs e)
    {
        decimal salary = 0;
        decimal.TryParse(this.Salary.Text, out salary);

        this.Job = CreateJob(this.JobTitle.Text, salary);
        InitializeControls();
    }

    protected void Update_Click(object sender, EventArgs e)
    {
        decimal salary = 0;
        decimal.TryParse(this.Salary.Text, out salary);

        this.Job = CreateJob(this.JobTitle.Text, salary);
        InitializeControls();
    }
}
```

After filling in a job title, salary, and clicking Create Job, the page is rendered in the browser, as shown in Figure 9-14.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



Licensed by Matt Allen 21755280

Figure 9-14. *The page rendered in a browser after the new job has been created*

How It Works

We used the DataContractSerializer to serialize the self-tracking entity into a byte array. Once in the byte array, we can assign it to the ViewState. To get the self-tracking entity from the ViewState, we read the byte array and deserialize the entity. We needed to use the DataContractSerializer because the current version of self-tracking entities does not support the binary serialization. A future version would likely support it.

Binary serialization works for POCO entities as long as they are marked with the Serializable attribute. Dynamic proxies can also be serialized and deserialized when using binary serialization. However, when deserializing the proxy, if the proxy is not already loaded in the AppDomain, you will get an exception. To avoid the exception, you can force the proxy to be loaded in the AppDomain before deserialization by calling `CreateProxyTypes()`. You don't need to call `CreateProxyTypes()` if the serialization and de-serialization occur in the same AppDomain because the initial retrieval of the entity would have created the proxy in the AppDomain.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

9-10. Fixing Duplicate References on a WCF Client

Problem

You have made several calls to a WCF service. The client has obtained two references from the service to the same object. You want to fix the client-side graph so that it has only one reference to the object.

Solution

If the client's object graph contains two references to the same object, any attempt at saving the graph will throw a duplicate key exception on the service side. Duplicate references must be removed from the graph before it can be saved on the service side.

Suppose you have the database tables shown Figure 9-15.

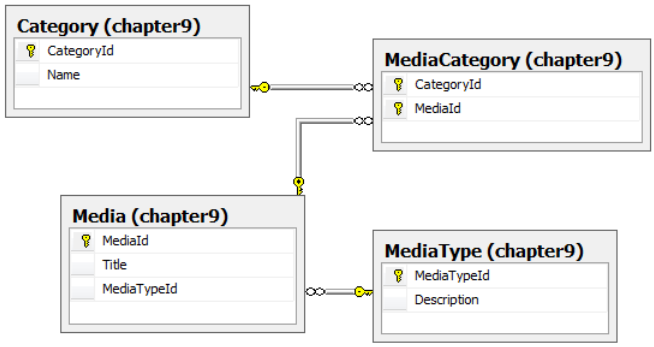


Figure 9-15. *A database diagram with tables and relationships representing media in categories*

To construct an example of how duplicate references can be created and fixed in the client's graph, do the following:

1. Create a new blank solution. Right-click the solution and select Add ➤ New Project. Select Windows Class Library. Name the project **MediaData** and click OK.
2. Right-click the project and add a new ADO.NET Data Model. Import the tables in Figure 9-15 into the model. The model should look like the one shown in Figure 9-16.
3. Right-click the design surface and select Add Code Generation Item. Select the ADO.NET Self-Tracking Entity Generator template under the Code templates. Name the template **MediaTemplate.tt**.
4. Right-click the solution and select Add ➤ New Project. Select Windows Class Library. Name the new library **MediaEntities**. Add a reference in this project to System.Runtime.Serialization.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5. Move the MediaTemplate.tt file from the MediaData project to the MediaEntities project. Because we've moved the template, we need to update its reference to the .edmx file. Edit the MediaTemplate.tt file and change the inputFile variable to point to the .edmx file in the MediaData project. If you've named your file Recipe10.edmx, the template file's inputFile should be set as follows:

```
string inputFile = @"../MediaData/Recipe10.edmx";
```

6. Add a reference on the MediaData project to the MediaEntities project.
7. Right-click the MediaTemplateContext.tt file and view its properties. Set the Custom Tool Namespace to MediaEntities. The template will now generate the object context in the same namespace as the entities.
8. Right-click the solution and select Add ➤ New Project. Select WCF Service Application from the WCF templates. Name this new service **MediaServices**. Add references to the MediaData and MediaEntities projects. Add a reference to System.Data.Entity.
9. Copy the <connectionString> section from the App.Config file in the MediaData project to the web.config file in the MediaServices project.
10. Change the service interface in the MediaServices project (in the file IService1.cs) to the code in Listing 9-34.
11. Change the implementation of the interface in the MediaServices project (in the file Service1.svc.cs) to the code in Listing 9-35. Make sure you add a using for System.Data.Entity.
12. Right-click the solution and select Add ➤ New Project. Select a Windows Console Application. Name the application **TestClient**.
13. Right-click the TestClient project and select Add Service Reference. Click Discover and select Service1 to add a service reference to our service. Add a reference to the MediaEntities project.
14. Use the code in Listing 9-36 to implement the TestClient. This code checks if the graph has two references to the same object and repairs the graph so that it can be sent to the service.

Figure 9-16. *A model for representing media in categories*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 9-34. The interface IService1 for our service

```
[ServiceContract]
public interface IService1
{
    [OperationContract]
    void Initialize();

    [OperationContract]
    Medium GetMediaByTitle(string title);

    [OperationContract]
    void SubmitCategory(Category category);
}
```

Listing 9-35. The implementation of the IService1 interface

```
public class Service1 : IService1
{
    public void Initialize()
    {
        using (var context = new EFRecipesEntities())
        {
            // clean up
            context.ExecuteStoreCommand("delete from chapter9.mediacategory");
            context.ExecuteStoreCommand("delete from chapter9.category");
            context.ExecuteStoreCommand("delete from chapter9.media");
            context.ExecuteStoreCommand("delete from chapter9.mediatype");

            // insert some test data
            var mediaType = new MediaType { MediaTypeId = 1,
                                            Description = "Article" };
            var media1 = new Medium { Title = "How to Design a Brick Fireplace",
                                      MediaType = mediaType };
            var media2 = new Medium { Title = "Repairing a Brick Oven",
                                      MediaType = mediaType };
            context.Media.AddObject(media1);
            context.Media.AddObject(media2);
            context.SaveChanges();
        }
    }

    public Medium GetMediaByTitle(string title)
    {
        using (var context = new EFRecipesEntities())
        {
            return context.Media.Include("MediaType")
                                .First(m => m.Title == title);
        }
    }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    public void SubmitCategory(Category category)
    {
        using (var context = new EFRecipesEntities())
        {
            context.Categories.ApplyChanges(category);
            context.SaveChanges();
        }
    }
}
```

Listing 9-36. The implementation of our TestClient

```
class Program
{
    static void Main(string[] args)
    {
        var service = new Service1Client();
        service.Initialize();
        var media1 = service.GetMediaByTitle("How to Design a Brick Fireplace");
        var media2 = service.GetMediaByTitle("Repairing a Brick Oven");
        var category = new Category { Name = "Brick Construction" };
        category.Media.Add(media1);
        category.Media.Add(media2);

        // at this point, media1 and media2 both
        // reference the "Article" MediaType, but there
        // are two instances of this object each with
        // the same key, we need to fix this
        if (media1.MediaType.MediaTypeId == media2.MediaType.MediaTypeId)
        {
            // apply fix
            media2.StopTracking();
            media2.MediaType.StopTracking();
            media1.MediaType.StopTracking();
            media2.MediaType = media1.MediaType;
            media2.StartTracking();
            media2.MediaType.StartTracking();
        }
        service.SubmitCategory(category);
    }
}
```

How It Works

On the client side, the two media objects have the same media type. Because these two objects were retrieved from the service by two separate calls, each media references its own instance of the media type. The object graph on the client side is invalid. It contains two instances of an entity, each with the same key. If we were to send this graph to the service, the service would throw an InvalidOperationException when it attempts to use the graph. In our case, the **AcceptChanges()** call on the service side throws the exception.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

There are a few different approaches we can take to avoid sending an invalid object graph to the service.

The first approach is to make sure that each object graph sent to the service was constructed entirely from objects received from a previous service call. Because an object graph received by the client is known to be valid, it is safe to send the same graph back to the service.

If the client can detect that two or more objects reference duplicate objects, then it can elect to send each of the valid subgraphs separately to the service. In our example, we could have added media1 to the category and then called **SubmitCategory()** to send the subgraph to the service. This would be followed by adding media2 to the category and calling **SubmitCategory()** again.

If the client can detect that an object graph is invalid, it can repair the graph, as we have in the code in Listing 9-36. Our graph was simple enough that it was easy to detect an invalid graph and easy to repair it.

Perhaps the most convenient approach is to not modify the graph using reference types, but to set the foreign key values. In our case, we can avoid sending instances of MediaType with the medium entities. The client would have to set the MediaTypeId values directly.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 10

Stored Procedures

Stored procedures are fixtures in the life of just about anyone who uses modern relational database systems such as Microsoft's SQL Server. A *stored procedure* is a bit of code that lives on the database server and often acts as an abstraction layer isolating the code consuming the data from many of the details of the physical organization of the data. Stored procedures can increase performance by moving data-intensive computations closer to the data and can act as a data-side repository for business and security logic. The bottom line is that if you use data, you will at some point consume it through a stored procedure.

In this chapter, we explore a number of recipes specifically focused around using stored procedures with Entity Framework. We used stored procedures in other recipes throughout this book, but usually they were in the context of implementing Insert, Update, and Delete actions. In this chapter, we'll show you several ways to consume the data exposed by stored procedures.

10-1. Returning an Entity Collection

Problem

You want to get an entity collection from a stored procedure.

Solution

Let's say you have a model like the one in Figure 10-1.

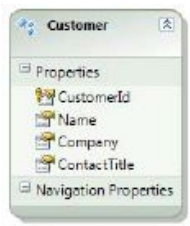


Figure 10-1. *A simple model of a customer*

In the database we have defined the stored procedure in Listing 10-1 that returns customers for given a company name and customer title.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 10-1. GetCustomers returns all the customers with the given title in the given company

```
create procedure Chapter10.GetCustomers
(@Company varchar(50),@ContactTitle varchar(50))
as
begin
        select * from
        chapter10.Customer where
        (@Company is null or Company = @Company) and
        (@ContactTitle is null or ContactTitle = @ContactTitle)
End
```

To use the GetCustomers stored procedure in the model, do the following.

1. Right-click the design surface and select Update Model From Database. In the dialog box, select the GetCustomers stored procedure. Click Finish to add the stored procedure to the model.
2. Right-click the design surface and select Add ➤Function Import. Select the GetCustomers stored procedure from the Stored Procedure Name drop-down. In the Function Import Name text box, enter **GetCustomers**. This will be the name used for the method in the model. Select the Entities Return Type and select Customer in the drop-down. Then click OK (see Figure 10-2).
3. Follow the pattern in Listing 10-2 to use the GetCustomers stored procedure.

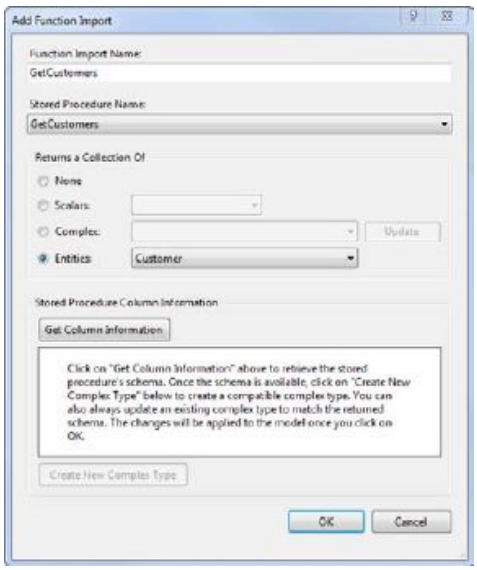


Figure 10-2. *The Add Function Import dialog box. Select the GetCustomers stored procedure, name the model function* ***GetCustomers****, and set the return type to Entities.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 10-2. Querying the model with the GetCustomers stored procedure via the ***GetCustomers()*** *method*

```
using (var context = new EFRecipesEntities())
{
    var c1 = new Customer {Name = "Robin Steele", Company = "GoShopNow.com",
                           ContactTitle="CEO"};
    var c2 = new Customer {Name = "Orin Torrey", Company = "GoShopNow.com",
                           ContactTitle="Sales Manager"};
    var c3 = new Customer {Name = "Robert Lancaster", Company = "GoShopNow.com",
                           ContactTitle = "Sales Manager"};
    var c4 = new Customer { Name = "Julie Stevens", Company = "GoShopNow.com",
                           ContactTitle = "Sales Manager" };
    context.Customers.AddObject(c1);
    context.Customers.AddObject(c2);
    context.Customers.AddObject(c3);
    context.Customers.AddObject(c4);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var allCustomers = context.GetCustomers("GoShopNow.com", "Sales Manager");
    Console.WriteLine("Customers that are Sales Managers at GoShopNow.com");
    foreach (var c in allCustomers)
    {
        Console.WriteLine("Customer: {0}", c.Name);
    }
}
```

The following is the output of the code in Listing 10-2:

```
Customers that are Sales Managers at GoShopNow.com

Customer: Orin Torrey

Customer: Robert Lancaster

Customer: Julie Stevens
```

How It Works

To retrieve an entity collection from a stored procedure in the database, we updated the model with the stored procedure from the database. Updating the model with the stored procedure added the code in Listing 10-3 to the SSDL section in the .edmx file.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 10-3. SSDL section updated with the GetCustomers stored procedure reference

```
<Function Name="GetCustomers" Aggregate="false" BuiltIn="false"
          NiladicFunction="false" IsComposable="false"
          ParameterTypeSemantics="AllowImplicitConversion" Schema="Chapter10">
  <Parameter Name="Company" Type="varchar" Mode="In" />
  <Parameter Name="ContactTitle" Type="varchar" Mode="In" />
</Function>
```

Next, we added a Function Import and specified both the name of the stored procedure and the name we want to expose for it in the model. We also defined the return type as entities of type Customer. Importing the GetCustomers stored procedure added the code in Listing 10-4 to the CSDL (conceptual) section of the .edmx file. This essentially defines signature for the function.

The mapping between the conceptual definition and the storage layer definition is added by the Function Import process through the **`<FunctionImportMapping>`** directive in the mapping section of the .edmx file. This maps the name of the stored procedure represented in the SSDL section to the name we provided for it in the model.

*Listing 10-4. CSDL section updated with the **`GetCustomers()`** method*

```
<FunctionImport Name="GetCustomers" EntitySet="Customers"
                ReturnType="Collection(EFRecipesModel.Customer)">
  <Parameter Name="Company" Mode="In" Type="String" />
  <Parameter Name="ContactTitle" Mode="In" Type="String" />
</FunctionImport>
```

A database function or stored procedure is composable if the results can be filtered. Most database systems support composing functions but not stored procedures. In Listing 10-3, the IsComposable attribute is set to **`false`**.

10-2. Returning Output Parameters

Problem

You want to retrieve values from one or more output parameters of a stored procedure.

Solution

Let's say you have a model like the one in Figure 10-3.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

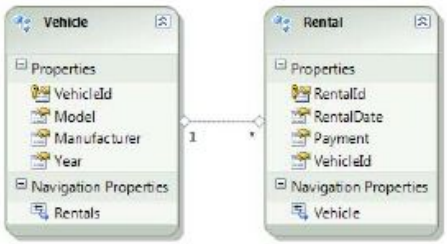


Figure 10-3. *A simple model for vehicle rental*

For a given date, you want to know the total number of rentals, the total rental payments made, and the vehicles rented. The stored procedure in Listing 10-5 is one way to get the information we want.

Licensed by Matt Allen 2175280

Listing 10-5. A stored procedure for the vehicles rented, the number of rentals, and the total rental payments

```
create procedure [chapter10].[GetVehiclesWithRentals]
(@date date,
@TotalRentals int output,
@TotalPayments decimal(18,2) output)
as
begin
  select @TotalRentals = COUNT(*), @TotalPayments = SUM(payment)
  from chapter10.Rental
  where RentalDate = @date

  select distinct v.*
  from chapter10.Vehicle v join chapter10.Rental r
  on v.VehicleId = r.VehicleId
end
```

To use the stored procedure in Listing 10-5 in the model, do the following.

1. Right-click the design surface and select Update Model From Database. In the dialog box, select the GetVehiclesWithRentals stored procedure. Click Finish to add the stored procedure to the model.
2. Right-click the design surface and select Add ➤ Function Import. Select the GetVehiclesWithRentals stored procedure from the Stored Procedure Name drop-down. In the Function Import Name text box, enter **GetVehiclesWithRentals**. This will be the name used for the method in the model. Select the Entities Return Type and select Vehicle in the drop-down. Click OK.
3. Follow the pattern in Listing 10-6 to use the GetVehiclesWithRentals stored procedure.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 10-6. Querying the model using the GetVehiclesWithRentals stored procedure via the ***GetVehiclesWithRentals()*** *method*

```
using (var context = new EFRecipesEntities())
{
    var car1 = new Vehicle { Manufacturer = "Toyota", Model = "Camry",
                             Year = 2010 };
    var car2 = new Vehicle { Manufacturer = "Chevrolet", Model = "Corvette",
                             Year = 2010 };
    var r1 = new Rental { Vehicle = car1,
                          RentalDate = DateTime.Parse("2/2/2010"),
                          Payment = 59.95M };
    var r2 = new Rental { Vehicle = car2,
                          RentalDate = DateTime.Parse("2/2/2010"),
                          Payment = 139.95M };
    context.AddToRentals(r1);
    context.AddToRentals(r2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    string reportDate = "2/2/2010";
    var totalRentals = new ObjectParameter("TotalRentals", typeof(int));
    var totalPayments = new ObjectParameter("TotalPayments", typeof(decimal));
    var vehicles = context.GetVehiclesWithRentals(DateTime.Parse(reportDate),
                     totalRentals, totalPayments);
    Console.WriteLine("Rental Activity for {0}",reportDate);
    Console.WriteLine("Vehicles Rented");
    foreach(var vehicle in vehicles)
    {
        Console.WriteLine("{0} {1} {2}",vehicle.Year.ToString(),
                          vehicle.Manufacturer, vehicle.Model);
    }
    Console.WriteLine("Total Rentals: {0}",
                      ((int)totalRentals.Value).ToString());
    Console.WriteLine("Total Payments: {0}",
                      ((decimal)totalPayments.Value).ToString("C"));
}
```

The following is the output of the code in Listing 10-6:

```
Rental Activity for 2/2/2010

Vehicles Rented

2010 Toyota Camry

2010 Chevrolet Corvette
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Total Rentals: 2

Total Payments: $200.00
```

How It Works

When we updated the model with the GetVehiclesWithRentals stored procedure, the wizard updated the store model with the stored procedure. By importing the function (in Step 2) we updated the conceptual model. The result is that the stored procedure is exposed as the **GetVehiclesWithRentals()** method, which has a signature semantically similar to the stored procedure.

There is one important thing to note when calling the **GetVehiclesWithRentals()** method: the returned entity collection must be materialized before the output parameters will become available. This should not be too surprising to those who have used multiple result sets in ADO.NET. The data reader must be advanced (with the **NextResult()** method) to the next result set. Similarly, the entire returned entity collection must be accessed or disposed before the output parameters can be accessed.

In our example, it is not enough to materialize the first vehicle for the output parameters to become available. The entire collection must be materialized. This means moving the lines printing the total rentals and total payments to a position after the **foreach** loop. Alternatively, we could materialize the entire collection with the **ToList()** method and then iterate through the list. This would allow us to access the output parameters prior to iterating through the collection.

10-3. Returning a Scalar Value Result Set

Problem

You want to use a stored procedure that returns a result set containing a single scalar value.

Solution

Let's say you have a model like the one in Figure 10-4.

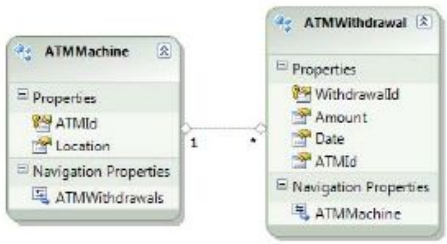


Figure 10-4. *A model representing ATM machines and withdrawal transactions*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

You want to use a stored procedure that returns the total amount withdrawn from a given ATM on a given date. The code in Listing 10-7 is one way to implement this stored procedure.

Listing 10-7. The GetWithdrawals stored procedure that returns the total amount withdrawn from a given ATM on a given date

```
create procedure [Chapter10].[GetWithdrawals]
(@ATMId int, @WithdrawalDate date)
as
begin
        select SUM(amount) TotalWithdrawals
        from Chapter10.ATMWithdrawal
        where ATMId = @ATMId and [date] = @WithdrawalDate
end
```

To use the stored procedure in Listing 10-7 in the model, do the following:

1. Right-click the design surface and select Update Model From Database. In the dialog box, select the GetWithdrawals stored procedure. Click Finish to add the stored procedure to the model.
2. Right-click the design surface and select Add ➤ Function Import. Select the GetWithdrawals stored procedure from the Stored Procedure Name drop-down. In the Function Import Name text box, enter GetWithdrawals. This will be the name used for the method in the model. Select the Scalars Return Type and select Decimal in the drop-down. Click OK.
3. Follow the pattern in Listing 10-8 to use the GetWithdrawals stored procedure.

Listing 10-8. Querying the model with the GetWithdrawals stored procedure via the ***GetWithdrawals()*** *method*

```
DateTime today = DateTime.Parse("2/2/2010");
DateTime yesterday = DateTime.Parse("2/1/2010");
using (var context = new EFRecipesEntities())
{
    var atm = new ATMMachine { ATMId = 17, Location = "12th and Main" };
    atm.ATMWithdrawals.Add(new ATMWithdrawal {Amount = 20.00M, Date= today});
    atm.ATMWithdrawals.Add(new ATMWithdrawal {Amount = 100.00M, Date = today});
    atm.ATMWithdrawals.Add(new ATMWithdrawal {Amount = 75.00M, Date = yesterday});
    atm.ATMWithdrawals.Add(new ATMWithdrawal {Amount = 50.00M, Date=  today});
    context.AddToATMMachines(atm);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var forToday = context.GetWithdrawals(17, today).FirstOrDefault();
    var forYesterday = context.GetWithdrawals(17, yesterday).FirstOrDefault();
    var atm = context.ATMMachines.Where(o => o.ATMId == 17).FirstOrDefault();
    Console.WriteLine("ATM Withdrawals for ATM at {0} at {1}",
             atm.ATMId.ToString(), atm.Location);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    Console.WriteLine("\t{0} Total Withdrawn = {1}",
            yesterday.ToShortDateString(), forYesterday.Value.ToString("C"));
    Console.WriteLine("\t{0} Total Withdrawn = {1}", today.ToShortDateString(),
            forToday.Value.ToString("C"));
}
```

The following is the output from the code in Listing 10-8:

```
ATM Withdrawals for ATM at 17 at 12th and Main

        2/1/2010 Total Withdrawn = $75.00

        2/2/2010 Total Withdrawn = $170.00
```

How It Works

Notice that Entity Framework expects the stored procedure to return a collection of scalar values. In our example, our store procedure returns just one decimal value. We use the `FirstOrDefault()` method to extract this scalar from the collection.

10-4. Returning a Complex Type from a Stored Procedure

Problem

You want to use a stored procedure that returns a complex type in the model.

Solution

Let's say you have a model with an Employee entity. Employee contains the employee's id, name, and a complex address type that holds the address, city, state, and ZIP code for the employee. The name of the complex type is EmployeeAddress. The property in the Employee entity is simply Address. The Employee entity is shown in Figure 10-5.

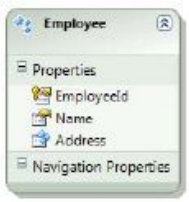


Figure 10-5. *An Employee entity with an Address property of type EmployeeAddress, which is a complex type*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

You want to use a stored procedure to return a collection of instances of the EmployeeAddress complex type. The stored procedure that returns the addresses might look like the one in Listing 10-9.

Listing 10-9. A stored procedure to return the addresses for employees in a given city

```
create procedure [Chapter10].[GetEmployeeAddresses]
(@city varchar(50))
as
begin
        select [address], city, [state], ZIP
        from Chapter10.Employee where city = @city
end
```

To use the stored procedure in Listing 10-9 in the model, do the following.

1. Right-click the design surface and select Update Model From Database. In the dialog box, select the GetEmployeeAddresses stored procedure. Click Finish to add the stored procedure to the model.
2. Right-click the design surface and select Add ➤ Function Import. Select the GetEmployeeAddresses stored procedure from the Stored Procedure Name drop-down. In the Function Import Name text box, enter **GetEmployeeAddresses**. This will be the name used for the method in the model. Select the Complex Return Type and select EmployeeAddress in the drop-down. Click OK.
3. Follow the pattern in Listing 10-10 to use the GetEmployeeAddresses stored procedure.

Listing 10-10. Querying the model using the GetEmployeeAddresses stored procedure via the ***GetEmployeeAddresses()*** *method*

```
using (var context = new EFRecipesEntities())
{
    var emp1 = new Employee { Name = "Lisa Jefferies",
                    Address = new EmployeeAddress {
                                 Address = "100 E. Main",
                                 City = "Fort Worth", State = "TX",
                                 ZIP = "76106" } };
    var emp2 = new Employee { Name = "Robert Jones",
                    Address = new EmployeeAddress {
                                Address = "3920 South Beach",
                                City = "Fort Worth", State = "TX",
                                ZIP = "76102" } };
    var emp3 = new Employee { Name = "Steven Chue",
                    Address = new EmployeeAddress {
                                Address = "129 Barker",
                                City = "Euless", State = "TX",
                                ZIP = "76092" } };
    var emp4 = new Employee { Name = "Karen Stevens",
                    Address = new EmployeeAddress {
                                Address = "108 W. Parker",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                City = "Fort Worth", State = "TX",
                                ZIP = "76102" } };
    context.AddToEmployees(emp1);
    context.AddToEmployees(emp2);
    context.AddToEmployees(emp3);
    context.AddToEmployees(emp4);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Employee addresses in Fort Worth, TX");
    foreach (var address in context.GetEmployeeAddresses("Fort Worth"))
    {
        Console.WriteLine("{0}, {1}, {2}, {3}", address.Address,
                          address.City, address.State, address.ZIP);
    }
}
```

The following is the output of the code in Listing 10-10:

```
Employee addresses in Fort Worth, TX

100 E. Main, Fort Worth, TX, 76106

3920 South Beach, Fort Worth, TX, 76102

108 W. Parker, Fort Worth, TX, 76102
```

How It Works

Complex types offer a convenient way to refactor repeated groups of properties into a single type that can be reused across many entities. In this recipe, we created a stored procedure that returned the address information for employees in a given city. In the model, we mapped these returned columns to the fields of the EmployeeAddress complex type. The **GetEmployeeAdresses()** method is defined by the Function Import Wizard to return a collection of instances of the EmployeeAddress type.

Complex types are often used to hold arbitrarily shaped data returned from a stored procedure. The data is not required to map to any entity in the model. Because complex types are not tracked by the object context, they are both a lightweight and efficient alternative to handling shaped data in the model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10-5. Defining a Custom Function in the Storage Model

Problem

You want to define a custom function inside the model rather than a stored procedure in the database.

Solution

Let's say you have a database that keeps track of members and the messages they have sent. Figure 10-6 shows one representation of this database.

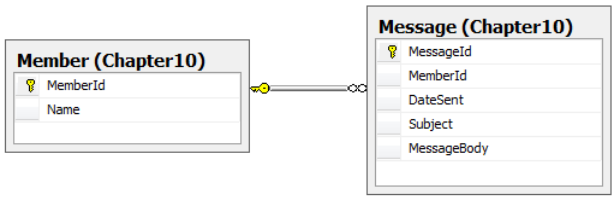


Figure 10-6. *A simple database of members and their messages*

Now it may be the case that, as a lowly programmer, you have not been granted access to the database to create stored procedures. However, being a wise and productive programmer, you want to encapsulate the query logic for finding the members with the highest number of messages into a reusable custom function in the storage model procedure. The model looks like the one in Figure 10-7.

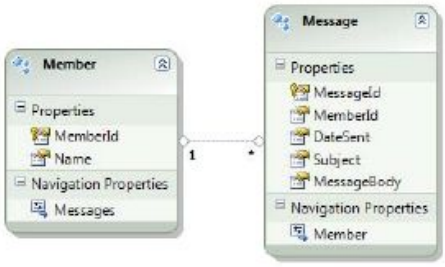


Figure 10-7. *The model for members and their messages*

To define the custom function in the storage model, do the following:

1. Right-click the .edmx file and select Open With ➤ XML Editor. This will open the .edmx file in the XML editor.
2. Add the code in Listing 10-11 into the `<Schema>` element. This defines the custom function.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3. Open the .edmx file in the Designer. Right-click the design surface and select Add ➤ Function Import. In the dialog box, select the MembersWithTheMostMessages in the Stored Procedure Name drop-down. Enter MembersWithTheMostMessages in the Function Import Name text box. Finally, select Entities as the return type and choose Member as the entity type. Click OK.
4. Follow the pattern in Listing 10-12 to use the **MembersWithTheMostMessages()** method, which exposes the MembersWithTheMostMessages custom function.

Listing 10-11. The definition of the custom function MembersWithTheMostMessages

```
<Function Name="MembersWithTheMostMessages" IsComposable="false">
  <CommandText>
    select m.*
    from chapter10.member m
    join
    (
    select distinct msg.MemberId
    from chapter10.message msg where datesent = @datesent
    ) temp on m.MemberId = temp.MemberId
  </CommandText>
  <Parameter Name="datesent" Type="date" />
</Function>
```

*Listing 10-12. Using the MembersWithTheMostMessages function via the **MembersWithTheMostMessages()** method*

```
DateTime today = DateTime.Parse("4/18/2010");
using (var context = new EFRecipesEntities())
{
    var mem1 = new Member { Name = "Jill Robertson" };
    var mem2 = new Member { Name = "Steven Rhodes" };
    mem1.Messages.Add(new Message { DateSent = today,
                                    MessageBody = "Hello Jim",
                                    Subject = "Hello" });
    mem1.Messages.Add(new Message { DateSent = today,
                                    MessageBody = "Wonderful weather!",
                                    Subject = "Weather" });
    mem1.Messages.Add(new Message { DateSent = today,
                                    MessageBody = "Meet me for lunch",
                                    Subject = "Lunch plans" });
    mem2.Messages.Add(new Message { DateSent = today,
                                    MessageBody = "Going to class today?",
                                    Subject = "What's up?" });
    context.Members.AddObject(mem1);
    context.Members.AddObject(mem2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
{
    Console.WriteLine("Members by message count for {0}",
                      today.ToShortDateString());
    var members = context.MembersWithTheMostMessages(today);
    foreach (var member in members)
    {
        Console.WriteLine("Member: {0}", member.Name);
    }
}
```

The following is the output of the code in Listing 10-12:

```
Members by message count for 4/18/2010

Member: Jill Robertson

Member: Steven Rhodes
```

How It Works

A custom function is different from a model defined function (see Chapter 11) in that a custom function is defined in the storage model. This makes the custom function much more like a traditional stored procedure in a database. Just like a DefiningQuery in the storage model defines a "virtual" table that doesn't really exist in the database, a custom function in the storage model is like a "virtual" stored procedure. Some in the Entity Framework community refer to custom functions as *native functions*. The Microsoft documentation uses the term *custom function*, so we'll go with that.

The code in Listing 10-11 defines our custom function. We put this in the storage model section of the .edmx file by directly editing the file using the XML editor. Note that if you use the Update From Database Wizard to update the model with new objects from your database, the wizard will overwrite this section. So, be careful to save out any changes you've made to the storage model before you use the Update From Database Wizard.

Just like with the stored procedures in the previous recipes, we used the Function Import Wizard to map the custom function to a CLR method. This defines the name of the CLR method and the expected return type. In our case, the Custom Function returns a collection of instances of the Member entity.

In Listing 10-12, the code uses the `MembersWithTheMostMessages()` method to invoke the custom function. This is the same pattern we used with stored procedures.

Custom functions can be helpful in the following scenarios:

- You don't have permissions to create the stored procedures you need in the database.
- You want to manage deployments of the code and the database separately. Using one or more custom functions, you can deploy your code without deploying new stored procedures for the database.
- The existing stored procedures in the database have parameters that are incompatible with your entities. Using custom functions, you can create an abstraction layer that drops, adds, or changes types between the stored procedure parameters and the properties on your entity.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10-6. Populating Entities in a Table per Type Inheritance Model

Problem

You want to use a stored procedure to populate entities in a Table per Type inheritance model.

Solution

Let's say the model looks like the one in Figure 10-8. In this model, the entities Magazine and DVD extend the base entity Media. In the underlying database, we have a table for each of these entities. We have modeled these tables using Table per Type inheritance. We want to use a stored procedure to obtain the data for this model from the database.

Licensed by Matt Allen 2175280

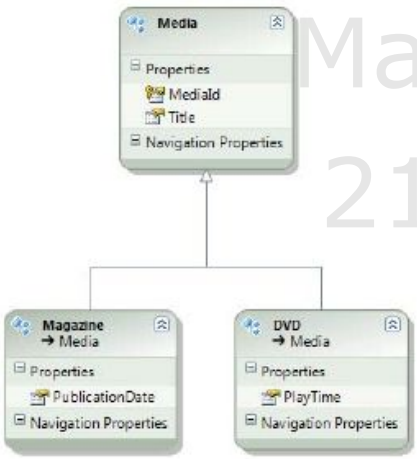


Figure 10-8. *A model using Table per Type inheritance. The model represents some information about magazines and DVDs.*

To create and use a stored procedure that returns these entities, do the following.

1. In your database, create the stored procedure in Listing 10-13.
2. Right-click the design surface and select Update Model from Database. Select the GetAllMedia stored procedure. Click Finish to add the stored procedure to the model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3. Right-click the design surface and select Add ➤ Function Import. In the dialog box, select the GetAllMedia stored procedure. Enter GetAllMedia in the Function Import Name text box. Select Entities as the type of collection and Media as the type of entity returned. Click OK. This will create the skeleton **<FunctionImportMapping>**.
4. Right-click the .edmx file and select Open With ➤ XML Editor. Edit the **<FunctionImportMapping>** tag in the mapping section of the .edmx file to match the code in Listing 10-14. This maps the rows returned by the stored procedure to either the Magazine or the DVD entity based on the MediaType column.
5. Follow the pattern in Listing 10-15 to use the GetAllMedia stored procedure via the **GetAllMedia()** method.

Listing 10-13. The GetAllMedia stored procedure that returns a rowset with a discriminator column

```
create procedure [Chapter10].[GetAllMedia]
as
begin
select m.MediaId,c.Title,m.PublicationDate, null PlayTime,'Magazine' MediaType
from chapter10.Media c join chapter10.Magazine m on c.MediaId = m.MediaId
union
select d.MediaId,c.Title,null,d.PlayTime,'DVD'
from chapter10.Media c join chapter10.DVD d on c.MediaId = d.MediaId
end
```

Listing 10-14. This FunctionImportMapping conditionally maps the returned rows to either the Magazine or the DVD entity.

```
<FunctionImportMapping FunctionImportName="GetAllMedia"
 FunctionName="EFRecipesModel.Store.GetAllMedia">
  <ResultMapping>
    <EntityTypeMapping TypeName="EFRecipesModel.Magazine">
      <ScalarProperty ColumnName="PublicationDate" Name="PublicationDate"/>
      <Condition ColumnName="MediaType" Value="Magazine"/>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="EFRecipesModel.DVD">
      <ScalarProperty ColumnName="PlayTime" Name="PlayTime"/>
      <Condition ColumnName="MediaType" Value="DVD"/>
    </EntityTypeMapping>
  </ResultMapping>
</FunctionImportMapping>
```

*Listing 10-15. Using the GetAllMedia stored procedure via the **GetAllMedia()** method*

```
using (var context = new EFRecipesEntities())
{
    context.MediaSet.AddObject(new Magazine { Title = "Field and Stream",
                        PublicationDate = DateTime.Parse("6/12/1945") });
    context.MediaSet.AddObject(new Magazine { Title = "National Geographic",
                        PublicationDate = DateTime.Parse("7/15/1976") });
    context.MediaSet.AddObject(new DVD { Title = "Harmony Road",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                        PlayTime = "2 hours, 30 minutes" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var allMedia = context.GetAllMedia();
    Console.WriteLine("All Media");
    Console.WriteLine("=========");
    foreach (var m in allMedia)
    {
        if (m is Magazine)
            Console.WriteLine("{0} Published: {1}", m.Title,
                              ((Magazine)m).PublicationDate.ToShortDateString());
        else if (m is DVD)
            Console.WriteLine("{0} Play Time: {1}", m.Title, ((DVD)m).PlayTime);
    }
}
```

The following is the output of the code in Listing 10-15:

```
All Media

=========

Field and Stream Published: 6/12/1945

National Geographic Published: 7/15/1976

Harmony Road Play Time: 2 hours, 30 minutes
```

How It Works

The two key parts to the solution are the discriminator column injected into the result set by the stored procedure and the conditional mapping of the results to the Magazine and DVD entities.

The stored procedure in Listing 10-13 forms a union of rows from the Magazine and DVD tables and injects the strings Magazine or DVD into the MediaType discriminator column. For each select, we join to the Media table, which is represented in the model by the base entity, to include the Title column. All the rows from all three tables are now in the result set with each row tagged to indicate the table it came from.

With each row tagged with either Magazine or DVD, we conditionally map the rows to either the Magazine or DVD entities, based on the tag or value in the discriminator column. This is done in the **<FunctionImportMapping>** section.

In Listing 10-15, we call the CLR method **GetAllMedia()**, which we mapped to the GetAllMedia stored procedure when we added the Function Import. When we call **GetAllMedia()**, the entire object graph is materialized with the inheritance hierarchy intact. We iterate through the collection, alternately printing out the Magazine and DVD entities.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10-7. Populating Entities in a Table per Hierarchy Inheritance Model

Problem

You want to use a stored procedure to populate entities in a Table per Hierarchy inheritance model.

Solution

Suppose you have a model like the one in Figure 10-9. We have two derived entities: Instructor and Student. Because this model is using Table per Hierarchy inheritance, we have just one table in the database. The Person table has a discriminator column that is used to map the table to the derived entities. You want to populate the entities with a stored procedure.

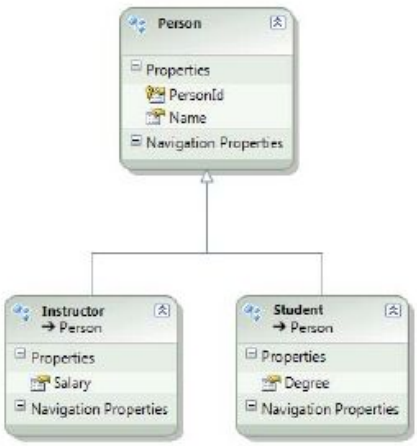


Figure 10-9. *A model for instructors and students*

To create and use a stored procedure that returns these entities, do the following:

1. In your database, create the stored procedure in Listing 10-16. This stored procedure returns all the people in the hierarchy.
2. Right-click the design surface and select Update Model from Database. Select the GetAllPeople stored procedure. Click Finish to add the stored procedure to the model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

3. Right-click the design surface and select Add ➤ Function Import. In the dialog box, select the GetAllPeople stored procedure. Enter GetAllPeople in the Function Import Name text box. Select Entities as the type of collection and Person as the type of entity returned. Click OK. This will create the skeleton **<FunctionImportMapping>** section.
4. Right-click the .edmx file and select Open With ➤ XML Editor. Edit the **<FunctionImportMapping>** tag in the mapping section of the .edmx file to match the code in Listing 10-17. This maps the rows returned by the stored procedure to either the Instructor or Student entity based on the PersonType column.
5. Follow the pattern in Listing 10-18 to use the GetAllPeople stored procedure via the **GetAllPeople()** method.

Listing 10-16. The GetAllPeople stored procedure; this stored procedure returns all the people, both Students and Instructors, in the model.

```
create procedure [Chapter10].[GetAllPeople]
as
begin
select * from chapter10.Person
end
```

Listing 10-17. The FunctionImportMapping conditionally maps rows to either the Instructor or Student entity

```
<FunctionImportMapping FunctionImportName="GetAllPeople"
        FunctionName="EFRecipesModel.Store.GetAllPeople">
  <ResultMapping>
    <EntityTypeMapping TypeName="EFRecipesModel.Student">
      <ScalarProperty Name="Degree" ColumnName="Degree" />
      <Condition ColumnName="PersonType" Value="Student"/>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="EFRecipesModel.Instructor">
      <ScalarProperty Name="Salary" ColumnName="Salary"/>
      <Condition ColumnName="PersonType" Value="Instructor"/>
    </EntityTypeMapping>
  </ResultMapping>
</FunctionImportMapping>
```

*Listing 10-18. Querying the model using the GetAllPeople stored procedure via the **GetAllPeople()** method*

```
using (var context = new EFRecipesEntities())
{
    context.People.AddObject(new Instructor { Name = "Karen Stanford",
                                              Salary = 62500M });
    context.People.AddObject(new Instructor { Name = "Robert Morris",
                                              Salary = 61800M });
    context.People.AddObject(new Student { Name = "Jill Mathers",
                                           Degree = "Computer Science" });
    context.People.AddObject(new Student { Name = "Steven Kennedy",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                        Degree = "Math" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Instructors and Students");
    var allPeople = context.GetAllPeople();
    foreach (var person in allPeople)
    {
        if (person is Instructor)
            Console.WriteLine("Instructor {0} makes {1}/year",
                              person.Name,
                              ((Instructor)person).Salary.ToString("C"));
        else if (person is Student)
            Console.WriteLine("Student {0}'s major is {1}",
                              person.Name, ((Student)person).Degree);
    }
}
```

The following is the output of the code in Listing 10-18:

```
Instructors and Students

Instructor Karen Stanford makes $62,500.00/year

Instructor Robert Morris makes $61,800.00/year

Student Jill Mathers's major is Computer Science

Student Steven Kennedy's major is Math
```

How It Works

Using a stored procedure to populate entities in a Table per Hierarchy inheritance model turns out to be a little easier than for Table per Type (see Recipe 6). Here the stored procedure just selected all rows in the Person table. The PersonType column contains the discriminator value that we use in `<FunctionImportMapping>` in Listing 10-17 to conditionally map the rows to either the Student or the Instructor entities. In Recipe 6, the stored procedure had to create the column. In both recipes, the key part is the conditional mapping in the `<FunctionImportMapping>` tag.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10-8. Mapping the Insert, Update, and Delete Actions to Stored Procedures

Problem

You want to map the Insert, Update, and Delete actions to stored procedures.

Solution

Let's say you have a model with the Athlete entity in Figure 10-10. The underlying database has the Athlete table in Figure 10-11. You want to use stored procedures for the Insert, Update, and Delete actions.

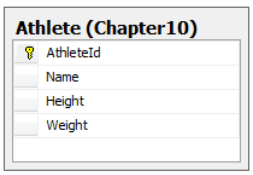


Figure 10-10. *The Athlete entity in the model*

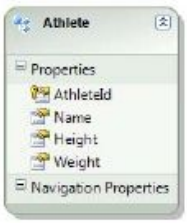


Figure 10-11. *The Athlete table with some basic information about athletes*

To map stored procedures to the Insert, Update, and Delete actions for the Athlete entity, do the following:

1. In your database, create the stored procedures in Listing 10-19.
2. Right-click the design surface and select Update Model from Database. Select the new stored procedures from Listing 10-19 and click Finish. This will add the stored procedures to the model.
3. Right-click the Athlete Entity and select Stored Procedure Mapping. Select the stored procedures for each of the actions. For the Insert action, map the return column AthleteId for the Insert action to the AthleteId property (see Figure 10-12).

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 10-19. The stored procedures for the Insert, Update, and Delete actions

```
create procedure [chapter10].[InsertAthlete]
(@Name varchar(50), @Height int, @Weight int)
as
begin
        insert into Chapter10.Athlete values (@Name, @Height, @Weight)
       select SCOPE_IDENTITY() as AthleteId
end
go

create procedure [chapter10].[UpdateAthlete]
(@AthleteId int, @Name varchar(50), @Height int, @Weight int)
as
begin
        update Chapter10.Athlete set Name = @Name, Height = @Height, [Weight] = @Weight
        where AthleteId = @AthleteId
end
go

create procedure [chapter10].[DeleteAthlete]
(@AthleteId int)
as
begin
        delete from Chapter10.Athlete where AthleteId = @AthleteId
end
```

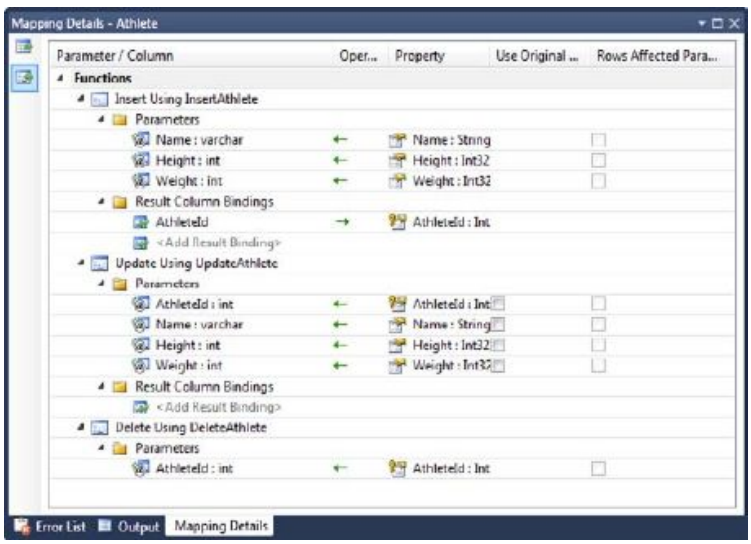


Figure 10-12. *Mapping the stored procedures, parameters, and return values for the Insert, Update, and Delete actions*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

We updated the model with the stored procedures we created in the database. This makes the stored procedures available for use in the model. Once we have the stored procedures available in the model, we mapped them to the Insert, Update, and Delete actions for the entity.

In this recipe, the stored procedures are about as simple as you can get. They take in properties as parameters and perform the action. For the Insert stored procedure, we need to return the stored generated key for the entity. In this recipe, the stored generated key is just an identity column. We need to return this from the stored procedure for the Insert action and map this returned value to the AthleteId property. This is an important step. Without this, Entity Framework would not be able to get the entity key for the instance of the Athlete entity just inserted.

When do I map stored procedures to the actions?

In most cases, Entity Framework will generate efficient code for the Insert, Update, and Delete actions. When would I ever need to replace this with my own stored procedures? Here are the best practice reasons why.

* Your company requires you to use stored procedures for some or all of the Insert, Update, or Delete activity for certain tables.
* You have additional tasks to do during one or more of the actions. For example, you might want to manage an audit trail or perform some complex business logic or perhaps leverage a user's privileges to execute stored procedures for security checking.
* Your entity is based on a QueryView (see Chapters 6 and 15) that requires you to map some or all of the actions to stored procedures.

The code in Listing 10-20 demonstrates inserting, deleting, and updating in the model. The code isn't any different because of the mapping of the actions. And that's fine. The fact that we have replaced the code that Entity Framework would have dynamically generated with our own stored procedures should not affect the code that uses the entity.

Listing 10-20. Executing the Insert, Update, and Delete actions

```
using (var context = new EFRecipesEntities())
{
    context.Athletes.AddObject(new Athlete { Name = "Nancy Steward",
                               Height = 167, Weight = 53 });
    context.Athletes.AddObject(new Athlete { Name = "Rob Achers",
                               Height = 170, Weight = 77 });
    context.Athletes.AddObject(new Athlete { Name = "Chuck Sanders",
                               Height = 171, Weight = 82 });
    context.Athletes.AddObject(new Athlete { Name = "Nancy Rodgers",
                               Height = 166, Weight = 59 });
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    // do a delete and an update
    var all = context.Athletes;
    context.DeleteObject(all.Where(o => o.Name == "Nancy Steward").First());
    all.Where(o => o.Name == "Rob Achers").First().Weight = 80;
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("All Athletes");
    Console.WriteLine("============");
    foreach (var athlete in context.Athletes)
    {
        Console.WriteLine("{0} weighs {1} Kg and is {2} cm in height",
         athlete.Name, athlete.Weight, athlete.Height);
    }
}
```

The following is the output of the code in Listing 10-20:

```
All Athletes

============

Rob Achers weighs 80 Kg and is 170 cm in height

Chuck Sanders weighs 82 Kg and is 171 cm in height

Nancy Rodgers weighs 59 Kg and is 166 cm in height
```

10-9. Using Stored Procedures for the Insert and Delete Actions in a Many-to-Many Association

Problem

You want to use stored procedures for the Insert and Delete actions in a payload-free, many-to-many association. These stored procedures affect only the link table in the association, not the associated entities.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have a many-to-many relationship between an Author table and a Book table. The link table, AuthorBook, is used as part of the relationship. See Figure 10-13.

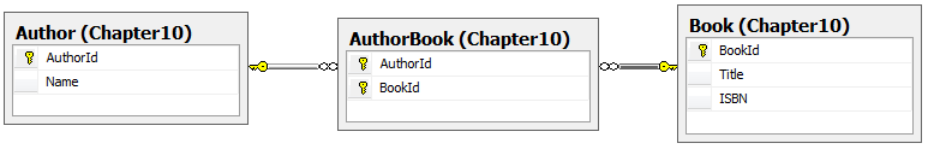


Figure 10-13. *A payload-free, many-to-many relationship between an Author and a Book*

When you import these tables into a model, you get a model that looks like the one in Figure 10-14.

Figure 10-14. *The model created by importing the tables in Figure 10-13*

Licensed by Matt Allen 2175280

To use stored procedures for the Insert and Delete actions, do the following.

1. In your database, create the stored procedures in Listing 10-21.
2. Right-click the design surface and select Update Model from Database. Select the new stored procedures from Listing 10-21 and click Finish. This will add the stored procedures to the model.
3. The current release of Entity Framework does not have designer support for mapping stored procedures to the Insert and Delete actions for an association. To perform this mapping manually, right-click the .edmx file and select Open With ➤ XML Editor. Add the code in Listing 10-22 in the Mappings section inside the `<AssociationSetMapping>` tag.

Listing 10-21. The stored procedures for the Insert and Delete actions

```
create procedure [chapter10].[InsertAuthorBook]
(@AuthorId int,@BookId int)
as
begin
        insert into chapter10.AuthorBook(AuthorId,BookId) values (@AuthorId,@BookId)
end
go
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
create proc [chapter10].[DeleteAuthorBook]
(@AuthorId int,@BookId int)
as
begin
        delete chapter10.AuthorBook where AuthorId = @AuthorId and BookId = @BookId
end
```

Listing 10-22. Mapping the stored procedures to the Insert and Delete actions for the many-to-many association

```
<ModificationFunctionMapping>
  <InsertFunction FunctionName="EFRecipesModel.Store.InsertAuthorBook">
    <EndProperty Name="Author">
      <ScalarProperty Name="AuthorId" ParameterName="AuthorId"  />
    </EndProperty>
    <EndProperty Name="Book">
      <ScalarProperty Name="BookId" ParameterName="BookId" />
    </EndProperty>
  </InsertFunction>
  <DeleteFunction FunctionName="EFRecipesModel.Store.DeleteAuthorBook">
    <EndProperty Name="Author">
      <ScalarProperty Name="AuthorId" ParameterName="AuthorId"  />
    </EndProperty>
    <EndProperty Name="Book">
      <ScalarProperty Name="BookId" ParameterName="BookId" />
    </EndProperty>
  </DeleteFunction>
</ModificationFunctionMapping>
```

The code in Listing 10-23 demonstrates inserting into and deleting from the model. As you can see from the SQL Profiler output that follows, our InsertAuthorBook and DeleteAuthorBook stored procedures are called when Entity Framework updates the many-to-many association.

Listing 10-23. Inserting into the model

```
using (var context = new EFRecipesEntities())
{
    var auth1 = new Author { Name = "Jane Austin"};
    var book1 = new Book { Title = "Pride and Prejudice",
                           ISBN = "1848373104" };
    var book2 = new Book { Title = "Sense and Sensibility",
                           ISBN = "1440469563" };
    auth1.Books.Add(book1);
    auth1.Books.Add(book2);
    var auth2 = new Author { Name = "Audrey Niffenegger" };
    var book3 = new Book { Title = "The Time Traveler's Wife",
                           ISBN = "015602943X" };
    auth2.Books.Add(book3);
    context.Authors.AddObject(auth1);
    context.Authors.AddObject(auth2);
    context.SaveChanges();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.DeleteObject(book1);
    context.SaveChanges();
}
```

Here is the output of the SQL Profiler showing the SQL statements that are executed by the code in Listing 10-23:

```
exec sp_executesql N'insert [Chapter10].[Author]([Name])

values (@0)

select [AuthorId]

from [Chapter10].[Author]

where @@ROWCOUNT > 0 and [AuthorId] = scope_identity()',N'@0 varchar(50)',

 @0='Jane Austin'


exec sp_executesql N'insert [Chapter10].[Author]([Name])

values (@0)

select [AuthorId]

from [Chapter10].[Author]

where @@ROWCOUNT > 0 and [AuthorId] = scope_identity()',N'@0 varchar(50)',

 @0='Audrey Niffenegger'


exec sp_executesql N'insert [Chapter10].[Book]([Title], [ISBN])

values (@0, @1)

select [BookId]

from [Chapter10].[Book]

where @@ROWCOUNT > 0 and [BookId] = scope_identity()',N'@0 varchar(50),

 @1 varchar(50)',@0='Pride and Prejudice',@1='1848373104'
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
exec sp_executesql N'insert [Chapter10].[Book]([Title], [ISBN])
values (@0, @1)
select [BookId]
from [Chapter10].[Book]
where @@ROWCOUNT > 0 and [BookId] = scope_identity()',N'@0 varchar(50),
 @1 varchar(50)',@0='Sense and Sensibility',@1='1440469563'

exec sp_executesql N'insert [Chapter10].[Book]([Title], [ISBN])
values (@0, @1)
select [BookId]
from [Chapter10].[Book]
where @@ROWCOUNT > 0 and [BookId] = scope_identity()',N'@0 varchar(50),
 @1 varchar(50)',@0='The Time Traveler''s Wife',@1='015602943X'

exec [Chapter10].[InsertAuthorBook] @AuthorId=1,@BookId=1

exec [Chapter10].[InsertAuthorBook] @AuthorId=1,@BookId=2

exec [Chapter10].[InsertAuthorBook] @AuthorId=2,@BookId=3

exec [Chapter10].[DeleteAuthorBook] @AuthorId=1,@BookId=1

exec sp_executesql N'delete [Chapter10].[Book]
 where ([BookId] = @0)',N'@0 int',@0=7
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

To map the stored procedures to the Insert and Delete actions for the many-to-many association, we created the stored procedures in our database then updated the model with the stored procedures.

Because Entity Framework's designer does not currently support mapping stored procedures to the Insert and Delete actions for associations, we need to edit the .edmx file directly. In the Mappings section, we added a `<ModificationFunctionMapping>` tag that maps the Insert and Delete actions for the association to our stored procedures. In this, we refer to the InsertAuthorBook and DeleteAuthorBook stored procedures which are defined in the Store model because we updated the model with these stored procedures from the database.

In the trace from Listing 10-23, we can see not only the expected inserts for the Author and Book table, but we can also see that our stored procedures are used for to insert and delete the association.

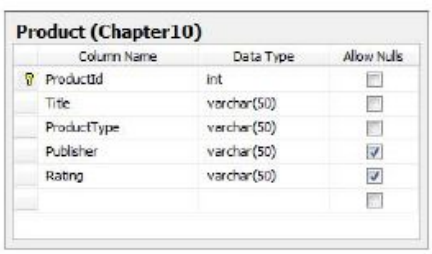
10-10. Mapping the Insert, Update, and Delete Actions to Stored Procedures for Table per Hierarchy Inheritance

Problems

You have a model that uses Table per Hierarchy inheritance and you want to map the Insert, Update, and Delete actions to stored procedures.

Solution

Let's say you database contains a Product table that describes a couple of different kinds of products (see Figure 10-15). You have created a model with derived types for each of the product types represented in the Product table. The model looks like the one in Figure 10-16.



Product (Chapter10)

Column Name	Data Type	Allow Nulls
ProductId	int	☐
Title	varchar(50)	☐
ProductType	varchar(50)	☐
Publisher	varchar(50)	☑
Rating	varchar(50)	☑
		☐

Figure 10-15. *A Product table with a discriminator column, ProductType, that indicates the type of product described by the row in the table*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

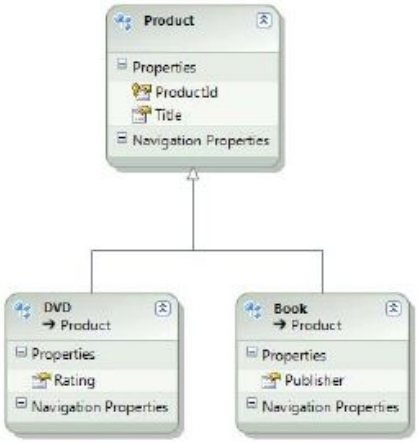


Figure 10-16. *A model using Table per Hierarchy inheritance with a derived type for each of the products*

To map stored procedures to the Insert, Update, and Delete actions for this model, do the following:

1. In your database, create the stored procedures in Listing 10-24. These stored procedures will handle the Insert, Update, and Delete actions for the Book and DVD entities.
2. Right-click the design surface and select Update Model from Database. Select the newly created stored procedures and click Finish to add them to the model.
3. Right-click the Book entity and select Stored Procedure Mapping. Map the InsertBook, UpdateBook, and DeleteBook stored procedures to the corresponding actions for the entity. Map the Result Column Binding for the Insert action to the ProductId property. See Figure 10-17.
4. Right-click the DVD entity and select Stored Procedure Mapping. Map the InsertDVD, UpdateDVD, and DeleteDVD stored procedures to the corresponding actions for the entity. Map the Result Column Binding for the Insert action to the ProductId property. See Figure 10-18.

Listing 10-24. The stored procedure we map to the Insert, Update, and Delete actions for the model

```
create procedure [chapter10].[InsertBook]
(@Title varchar(50), @Publisher varchar(50))
as
begin
        insert into Chapter10.Product (Title, Publisher, ProductType) values
           (@Title,@Publisher, 'Book')
        select SCOPE_IDENTITY() as ProductId
end
go
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
create procedure [chapter10].[UpdateBook]
(@Title varchar(50), @Publisher varchar(50), @ProductId int)
as
begin
        update Chapter10.Product set Title = @Title, Publisher = @Publisher
          where ProductId = @ProductId
end
go

create procedure [chapter10].[DeleteBook]
(@ProductId int)
as
begin
        delete from Chapter10.Product where ProductId = @ProductId
end
go

create procedure [chapter10].[InsertDVD]
(@Title varchar(50), @Rating varchar(50))
as
begin
        insert into Chapter10.Product (Title, Rating, ProductType) values
           (@Title, @Rating, 'DVD')
        select SCOPE_IDENTITY() as ProductId
end
go

create procedure [chapter10].[DeleteDVD]
(@ProductId int)
as
begin
        delete from Chapter10.Product where ProductId = @ProductId
end
go

create procedure [chapter10].[UpdateDVD]
(@Title varchar(50), @Rating varchar(50), @ProductId int)
as
begin
        update Chapter10.Product set Title = @Title, Rating = @Rating
          where ProductId = @ProductId
end
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

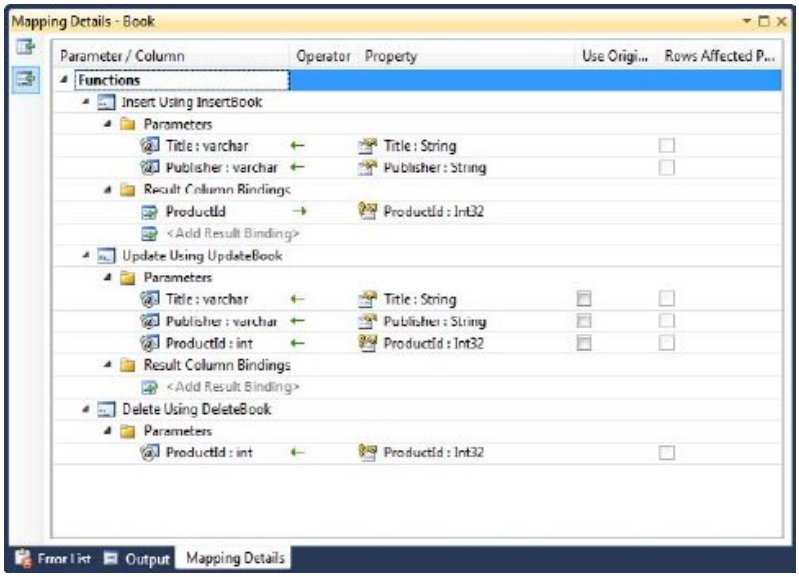


Figure 10-17. *Mapping the stored procedures to the Insert, Update, and Delete actions for the Book entity. Be particularly careful to map the Result Column Binding to the ProductId property for the Insert action.*

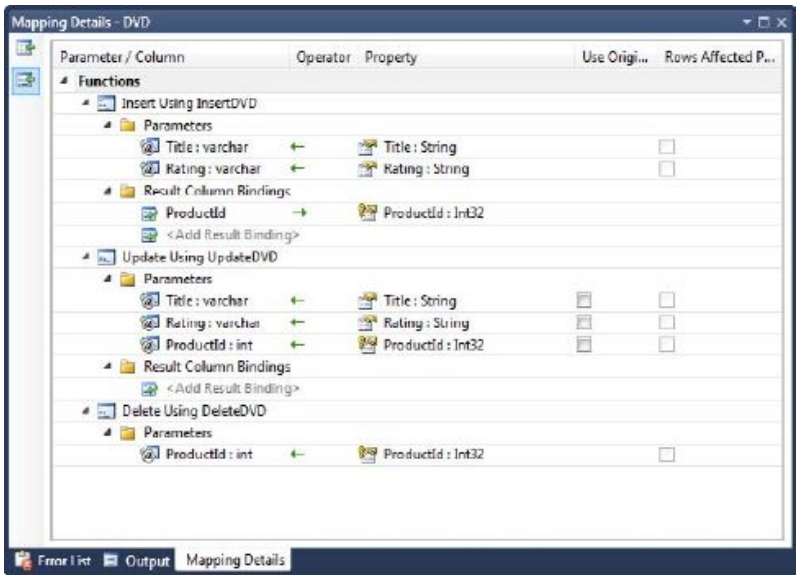


Figure 10-19. *Mapping the stored procedures to the Insert, Update, and Delete actions for the DVD entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

We created the stored procedures for the Insert, Update, and Delete actions for both the Book and DVD entities and imported them into the model. Once we have these stored procedures in the model, we mapped them to the corresponding actions, being careful to map the Result Column Binding for the Insert action to the ProductId property. This ensures that the store generated key for the Product is mapped to the ProductId property.

The Table per Hierarchy inheritance is supported by the implementation of the Insert stored procedures. Each of them inserts the correct ProductType value. Given these values in the tables, Entity Framework can correctly materialize the derived entities.

The code in Listing 10-25 demonstrates inserting, updating, deleting, and querying the model.

Listing 10-25. Exercising the Insert, Update, and Delete actions

```
using (var context = new EFRecipesEntities())
{
    var book1 = new Book { Title = "A Day in the Life",
                           Publisher = "Colorful Press" };
    var book2 = new Book { Title = "Spring in October",
                           Publisher = "AnimalCover Press" };
    var dvd1 = new DVD { Title = "Saving Sergeant Pepper", Rating = "G" };
    var dvd2 = new DVD { Title = "Around The Block", Rating = "PG-13" };
    context.Products.AddObject(book1);
    context.Products.AddObject(book2);
    context.Products.AddObject(dvd1);
    context.Products.AddObject(dvd2);
    context.SaveChanges();

    // update a book and delete a dvd
    book1.Title = "A Day in the Life of Sergeant Pepper";
    context.DeleteObject(dvd2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("All Products");
    Console.WriteLine("============");
    foreach (var product in context.Products)
    {
        if (product is Book)
            Console.WriteLine("'{0}' published by {1}",
                  product.Title, ((Book)product).Publisher);
        else if (product is DVD)
            Console.WriteLine("'{0}' is rated {1}",
                  product.Title, ((DVD)product).Rating);
    }
}
```

The following is the output of the code in Listing 10-25:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
All Products

============

'Spring in October' published by AnimalCover Press

'A Day in the Life of Sergeant Pepper' published by Colorful Press

'Saving Sergeant Pepper' is rated G
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 11

Functions

Functions provide a power mechanism for code reuse and a good way to make your code cleaner and more understandable. They can also be used to leverage code in the Entity Framework runtime as well as in the database layer.

In the first seven recipes we explore model defined functions. These functions are new in the current release of Entity Framework and allow you to create functions at the conceptual layer. These functions are defined in terms of Entity Framework types and your model entities. This makes them portable across data store implementations.

In the remaining recipes we show you how to use functions defined by Entity Framework and the database layer. These functions are implemented for you and allow you to leverage existing code either in Entity Framework's runtime or, closer to your data, in the database layer.

Licensed by Matt Allen 4173280

11-1. Returning a Scalar Value from a Model Defined Function

Problem

You want to define a function in the conceptual model that takes an instance of an entity and returns a scalar value.

Solution

Suppose you have a model like the one shown in Figure 11-1.

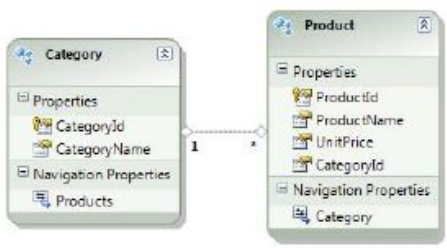


Figure 11-1. *A model for products and categories*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

To create a model defined function that takes an instance of the Category entity and returns the average unit price for all the products in the given category, do the following:

1. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor.
2. Insert the code in Listing 11-1 just below the **<Schema>** tag in the conceptual models section of the .edmx file. This defines the function in the model.
3. Insert into and query the model using code similar to pattern shown in Listing 11-2.

Listing 11-1. Definition of the AverageUnitPrice() function in the model

```
<Function Name="AverageUnitPrice" ReturnType="Edm.Decimal">
  <Parameter Name="category" Type="EFRecipesModel.Category" />
  <DefiningExpression>
    ANYELEMENT(Select VALUE Avg(p.UnitPrice)
    from EFRecipesEntities.Products as p where p.Category == category
    group by p.Category.CategoryName
    )
  </DefiningExpression>
</Function>
```

Listing 11-2. Inserting and querying the model using the model defined function AverageUnitPrice()

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var c1 = new Category { CategoryName = "Backpacking Tents" };
            new Product { ProductName = "Hooligan", UnitPrice = 89.99M,
                          Category = c1 };
            new Product { ProductName = "Kraz", UnitPrice = 99.99M,
                          Category = c1 };
            new Product { ProductName = "Sundome", UnitPrice = 49.99M,
                          Category = c1 };
            context.Categories.AddObject(c1);
            var c2 = new Category { CategoryName = "Family Tents" };
            new Product { ProductName = "Evanston", UnitPrice = 169.99M,
                          Category = c2 };
            new Product { ProductName = "Montana", UnitPrice = 149.99M,
                          Category = c2 };
            context.Categories.AddObject(c2);
            context.SaveChanges();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }
        // with eSQL
        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Using eSQL for the query...");
            Console.WriteLine();
            string sql = @"Select c.CategoryName, EFRecipesModel
                         .AverageUnitPrice(c) as AveragePrice from
                         EFRecipesEntities.Categories as c";
            var cats = context.CreateQuery<DbDataRecord>(sql);
            foreach (var cat in cats)
            {
                Console.WriteLine("Category '{0}' has an average price of {1}",
                                  cat[0], ((decimal)cat[1]).ToString("C"));
            }
        }

        // with LINQ
        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine();
            Console.WriteLine("Using LINQ for the query...");
            Console.WriteLine();
            var cats = from c in context.Categories
                       select new { Name = c.CategoryName,
                               AveragePrice = MyFunctions.AverageUnitPrice(c) };
            foreach (var cat in cats)
            {
                Console.WriteLine("Category '{0}' has an average price of {1}",
                                  cat.Name, cat.AveragePrice.ToString("C"));
            }
        }
    }
}

public class MyFunctions
{
    [EdmFunction("EFRecipesModel", "AverageUnitPrice")]
    public static decimal AverageUnitPrice(Category category)
    {
        throw new NotSupportedException("Direct calls are not supported!");
    }
}
```

The following is the output from the code in Listing 11-2:

```
Using eSQL for the query...
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Category 'Backpacking Tents' has an average price of $79.99

Category 'Family Tents' has an average price of $159.99



Using LINQ for the query...



Category 'Backpacking Tents' has an average price of $79.99

Category 'Family Tents' has an average price of $159.99
```

How It Works

Model defined functions are created in the conceptual layer and written in eSQL. Of course, this allows you to program against the entities in your model as we have done here, referencing the Category and Product entities and their association in the function's implementation. The added benefit is that we are not tied to a specific storage layer. We could swap out the lower layers, even the database provider, and our program would still work.

The designer currently provides no support for model defined functions. Unlike stored procedures, which are supported by the designer, model defined functions do not show up in the model browser nor anywhere else in the designer. The designer will not check for syntax errors in the eSQL. These you will find out about at runtime. However, the designer will at least tolerate model defined functions enough to open the .edmx file.

In Listing 11-2, the code starts off by inserting a couple of categories and a few products for each. Once we have the data in place, we query it using two slightly different approaches.

In the first query example, we build an eSQL statement that calls the **AverageUnitPrice()** function. We create and execute the query. For each row in the results, we pull out the data for the first column, which is the category name, and the data for the second column, which is the average unit price for the category. We display them for each row.

The second query example is a little more interesting. Here we use the **AverageUnitPrice()** function in a LINQ query. To do this, we need to add a stub method in a separate class. The method is decorated with the **[EdmFunction()]** attribute, which marks it as an implementation of a model defined function. This CLR method will not actually be called, which is evident by the exception we throw in the body of the method. Because we return a scalar value, the method's implementation here is simply for the signature (the parameter number, types, and return type). In the LINQ query, we grab each category and reshape the results into an anonymous type that holds the category name and the result of calling the **AverageUnitPrice()** method in the **MyFunction** class. This is the stub we created that is tied to the **AverageUnitPrice()** model defined function. For each of the resulting objects, we display the category name and the category's average unit price.

The parameters for model define functions can be scalar, entity types, complex types, anonymous types, or collections of these. In many of the recipes in this chapter, we'll show you how to create and use model defined functions with these parameter types.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The parameters for model defined functions don't show direction. There are no 'out' parameters, only implied 'in' parameters. The reason for this is that model defined functions are composable and can be used as part of LINQ queries. This prevents them from returning values in output parameters.

In this example, we returned a single scalar decimal value. To do this, we had to explicitly return a scalar using the AnyElement operator. Entity Framework does not know how to map a collection to a scalar value. We help out here by using the AnyElement operator, which signals that only a single value will result from the query. It just so happens that we return a collection of just one element from which the AnyElement operator selects just one element.

Best Practice

Model defined functions provide a clean and practical way to implement parts of a conceptual model that would be tedious if not impossible any other way. Here are some best practices and uses for model defined functions.

Model defined functions are written in eSQL and defined at the conceptual layer. This provides a level of abstraction from the details of the store layer and allows you to leverage a more complete model independent of the store layer.

You can define functions for expressions you commonly use in your LINQ or eSQL queries. This provides better code organization and allows code reuse.

Model defined functions are composable, which allows you to implement functions that serve as building blocks for more complex expressions. This can both simplify your code and make it more maintainable.

Model defined functions can be used in places where you have computed properties. A computed property, like a function, is a read-only value. For properties, you incur the cost of computing the value when the entity is materialized, whether or not you need the computed property. With a model defined function, the cost of computing the value is incurred only when you actually need the value.

11-2. Filtering an Entity Collection Using a Model Defined Function

Problem

You want to create a model defined function that filters a collection.

Solution

Suppose we have a model with Customers and Invoices, as shown in Figure 11-2.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

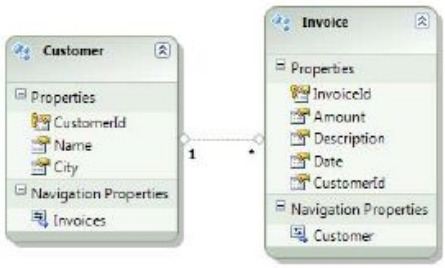


Figure 11-2. *Customer and Invoice in a model*

Let's say we want to create a model defined function that takes a collection of invoices and filters the collection to those invoices that have an amount greater than $300. Just for fun, let's use this model defined function in a query that further filters this collection to just those invoices created after 5/1/2009. Of course, we'll want to load all the customers associated with these invoices.

To get started, do the following:

1. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor.
2. Insert the code in Listing 11-3 just below the **<Schema>** tag in the conceptual models section of the .edmx file. This defines the function in the model.
3. Insert into and query the model using code similar to pattern shown in Listing 11-4.

Listing 11-3. The GetInvoices() model defined function

```
<Function Name="GetInvoices" ReturnType="Collection(EFRecipesModel.Invoice)" >
  <Parameter Name="invoices" Type="Collection(EFRecipesModel.Invoice)">
  </Parameter>
  <DefiningExpression>
    Select VALUE i
    from invoices as i where i.Amount > 300M
  </DefiningExpression>
</Function>
```

Listing 11-4. Querying the model using the GetInvoices() model defined function with both eSQL and LINQ

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    {
        DateTime d1 = DateTime.Parse("8/8/2009");
        DateTime d2 = DateTime.Parse("8/12/2008");
        var c1 = new Customer { Name = "Jill Robinson", City = "Dallas" };
        var c2 = new Customer { Name = "Jerry Jones", City = "Denver" };
        var c3 = new Customer { Name = "Janis Brady", City = "Dallas" };
        var c4 = new Customer { Name = "Steve Foster", City = "Dallas" };
        context.Invoices.AddObject(new Invoice { Amount = 302.99M,
            Description = "New Tires", Date = d1, Customer = c1 });
        context.Invoices.AddObject(new Invoice { Amount = 430.39M,
            Description = "Brakes and Shocks", Date = d1, Customer = c2 });
        context.Invoices.AddObject(new Invoice { Amount = 102.28M,
            Description = "Wheel Alignment", Date = d1, Customer = c3 });
        context.Invoices.AddObject(new Invoice { Amount = 629.82M,
            Description = "A/C Repair", Date = d2, Customer = c4 });
        context.SaveChanges();
    }

    using (var context = new EFRecipesEntities())
    {
        Console.WriteLine("Using eSQL query...");
        string sql = @"Select value i from
                EFRecipesModel.GetInvoices(EFRecipesEntities.Invoices) as i
                where i.Date > DATETIME'2009-05-1 00:00'
                and i.Customer.City = @City";
        var invoices = context.CreateQuery<Invoice>(sql,
               new ObjectParameter("City", "Dallas")).Include("Customer");
        foreach (var invoice in invoices)
        {
            Console.WriteLine("Customer: {0}\tInvoice for: {1}, Amount: {2}",
                 invoice.Customer.Name, invoice.Description, invoice.Amount);
        }
    }

    using (var context = new EFRecipesEntities())
    {
        Console.WriteLine();
        Console.WriteLine("Using LINQ query...");
        DateTime date = DateTime.Parse("5/1/2009");
        var invoices = from invoice in
                        MyFunctions.GetInvoices(context.Invoices)
                       where invoice.Date > date
                       where invoice.Customer.City == "Dallas"
                       select invoice;
        foreach (var invoice in ((ObjectQuery<Invoice>)invoices)
                                        .Include("Customer"))
        {
            Console.WriteLine("Customer: {0}, Invoice for: {1}, Amount: {2}",
                 invoice.Customer.Name, invoice.Description, invoice.Amount);
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
}

public class MyFunctions
{
    [EdmFunction("EFRecipesModel", "GetInvoices")]
    public static IQueryable<Invoice> GetInvoices(IQueryable<Invoice> invoices)
    {
        return invoices.Provider.CreateQuery<Invoice>(
            Expression.Call((MethodInfo)MethodInfo.GetCurrentMethod(),
                            Expression.Constant(invoices,
                                                typeof(IQueryable<Invoice>))));
    }
}
```

How It Works

From the definition of our `GetInvoices()` function in Listing 11-3, we see that it takes a collection of Invoices and returns a collection of Invoices. On the CLR side, this translates to taking an `IQueryable<Invoice>` and returning an `IQueryable<Invoice>`.

In the eSQLexpression we use the `GetInvoices()` function in the `from` clause. We pass in the unfiltered collection of invoices and our `GetInvoices()` function returns the filtered collection. We further filter the collection by date and the customer's city using a `where` clause. Then we use `CreateQuery<Invoice>()` to build the `ObjectQuery<Invoice>`. In building the query, we pass in the parameter to filter by city and use the `Include()` method to include the related customers. See the recipes in Chapter 5 for other examples of using the `Include()` method.

Once we have the `ObjectQuery<Invoice>`, we iterate over the resulting collection and print out the invoices that matched the two filters we applied.

For the LINQ query, the story is a little more interesting. Here we build the expression using the `GetInvoices()` method in the `from` clause and filter the resulting collection by date and city much like we did with the eSQLexpression. But to use our function in a LINQ query, we need to implement a CLR method that takes an `IQueryable<Invoice>` and returns an `IQueryable<Invoice>`. Unlike the stub method in Recipe 11-1, in which the model defined function returned a scalar value, here we have to provide an implementation in the body of the method. Creating this method is often referred to as *bootstrapping*.

Here are some rules for bootstrapping:

- Bootstrapping is required when a model defined function returns an `IQueryable<T>`.
- When a function returns an `IQueryable<T>`, but does not take an `IQueryable<T>`, the bootstrapping method must be implemented in a partial class of the ObjectContext.

The second rule comes about because we can't return an `IQueryable<T>` that has meaning in our ObjectContext without starting with an `IQueryable<T>`. If we pass in an `IQueryable<T>`, then we can perform some operation in our bootstrapping method that returns a related `IQueryable<T>`. However, we can't manufacture an `IQueryable<T>` outside of a partial class of our ObjectContext. In our example, we received an `IQueryable<T>` as a parameter, so we are free to implement the bootstrapping code outside of a partial class of our ObjectContext.

In the implementation of our bootstrapping method, we get an instance of `IQueryProvider` from the `IQueryable<Invoice>` through the Provider property. `IQueryProvider.CreateQuery<Invoice>()` allows us

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

to tack onto the expression tree for the `IQueryable<T>`. Here we add in the call to the `GetInvoices()` function, passing in the collection of invoices we have.

11-3. Returning a Computed Column from a Model Defined Function

Problem

You want to return a computed column from a model defined function.

Solution

Suppose we have an Employee entity containing the properties FirstName, LastName, and BirthDate, as shown in Figure 11-3.

Figure 11-3. *An Employee entity with a few typical properties*

We want to create a model defined function that returns the full name of the employee by combining the FirstName and LastName columns. We want to create another model defined function that returns the age of the employee based on the value in the BirthDate column.

To create and use these functions, do the following:

1. Right-click the .edmx file in the Solution Explorer and click Open With ➤ XML Editor. This will open the .edmx file in the XML Editor.
2. Insert the code in Listing 11-5 just below the `<Schema>` tag in the conceptual models section of the .edmx file. This defines the functions in the model.
3. Insert into and query the model using code similar to pattern shown in Listing 11-6.

Listing 11-5. Code for model defined functions

```
<Function Name="FullName" ReturnType="Edm.String">
  <Parameter Name="emp" Type="EFRecipesModel.Employee" />
  <DefiningExpression>
    Trim(emp.FirstName) + " " + Trim(emp.LastName)
  </DefiningExpression>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
</Function>
<Function Name="Age" ReturnType="Edm.Int32">
  <Parameter Name="emp" Type="EFRecipesModel.Employee" />
  <DefiningExpression>
    Year(CurrentDateTime()) - Year(emp.BirthDate)
  </DefiningExpression>
</Function>
```

Listing 11-6. Inserting into and querying the model invoking the model defined functions using both eSQL and LINQ

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            context.Employees.AddObject(new Employee { FirstName = "Jill",
               LastName = "Robins", Birthdate = DateTime.Parse("3/2/1976") });
            context.Employees.AddObject(new Employee { FirstName = "Michael",
               LastName = "Kirk", Birthdate = DateTime.Parse("4/12/1985") });
            context.Employees.AddObject(new Employee { FirstName = "Karen",
               LastName = "Stanford", Birthdate = DateTime.Parse("6/18/1963") });
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Query using eSQL");
            var esql = @"Select EFRecipesModel.FullName(e) as Name,
                         EFRecipesModel.Age(e) as Age from
                         EFRecipesEntities.Employees as e";
            var emps = context.CreateQuery<DbDataRecord>(esql);
            foreach (var emp in emps)
            {
                Console.WriteLine("Employee: {0}, Age: {1}", emp["Name"],
                                  emp["Age"]);
            }
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("\nQuery using LINQ");
            var emps = from e in context.Employees
                       select new
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                    {
                        Name = MyFunctions.FullName(e),
                        Age = MyFunctions.Age(e)
                    };
            foreach (var emp in emps)
            {
                Console.WriteLine("Employee: {0}, Age: {1}", emp.Name,
                                  emp.Age.ToString());
            }
        }
    }
}

public class MyFunctions
{
    [EdmFunction("EFRecipesModel", "FullName")]
    public static string FullName(Employee employee)
    {
        throw new NotSupportedException("Direct calls are not supported.");
    }

    [EdmFunction("EFRecipesModel", "Age")]
    public static int Age(Employee employee)
    {
        throw new NotSupportedException("Direct calls are not supported.");
    }
}
```

Licensed by Matt Allen 2175280

The output of the code in Listing 11-6 is the following:

```
Query using eSQL

Employee: Jill Robins, Age: 33

Employee: Michael Kirk, Age: 24

Employee: Karen Stanford, Age: 46


Query using LINQ

Employee: Jill Robins, Age: 33

Employee: Michael Kirk, Age: 24

Employee: Karen Stanford, Age: 46
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Our model defined functions return types **Edm.String** for the **FullName()** function and **Edm.Int32** for the **Age()** function. These functions are defined on the conceptual level so they don't directly refer to any type system outside of the Entity Data Model's type system. These primitive types are easily translated to the CLR type system.

In the **<DefiningExpression>** or body of the model defined functions, we directly access the properties of the entities we received in the parameters. There is no need to use a **select** statement. However, the resulting expression must have a type that matches the type defined as the return type of the function.

After inserting a few employees into our model, we first query using eSQL. We construct an eSQL expression that invokes our two model defined functions and projects the results to the Name and Age columns. Our eSQL expression results in a collection of anonymous types that contain just the Name and Age members. Because we're not returning one of the types defined in the model, we declare the type in **CreateQuery<T>()** to be **DbDataRecord**. We iterate over the collection resulting from the evaluation of the query and print out the employees' names and ages.

For the LINQ query, we select from the Employees entity set and project onto an anonymous type containing the Name and Age members. We set these members to the result of invoking our **FullName()** and **Age()** functions. As shown in the previous recipes in this chapter, we need to define the corresponding CLR methods. Because we are returning scalar values, these methods are never called and are used only for their signatures. The implementation of these methods reflects this.

We could have created read-only properties in a partial declaration of our Employee entity to implement the full name and age calculations. However, this would force the evaluation of these methods each time the entity is retrieved. With model defined functions, we perform the calculations only when needed.

11-4. Calling a Model Defined Function from a Model Defined Function

Problem

You want to use a model defined function in the implementation of another model defined function.

Solution

Suppose we have the model shown in Figure 11-4 representing the types of associates in a company along with their reporting structure.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

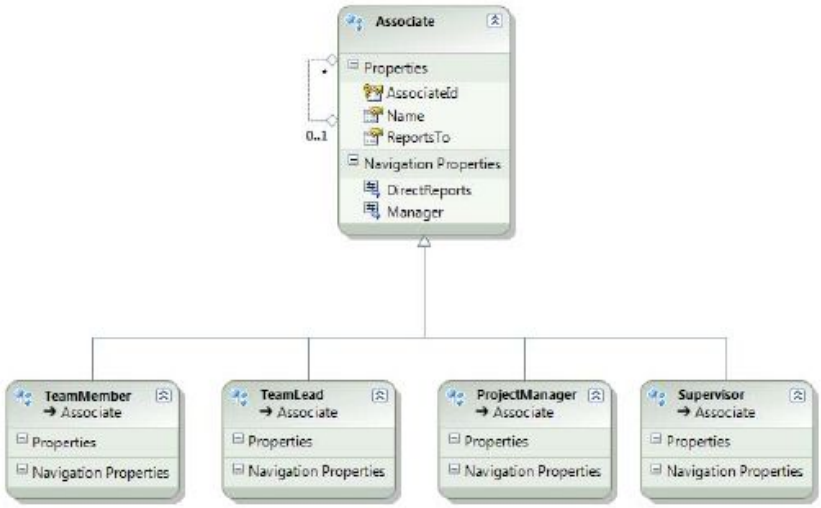


Figure 11-4. *A model representing the associate types in a company together with the reporting association*

In our fictional company, team members are managed by a team leader. Team leaders are managed by project managers. Supervisors manage the project managers. Of course, there could be many other associate types but for simplicity we'll stick with just these few.

If we wanted to return all the team members for a given project manager or supervisor, we would need to drill down through the project managers and team leaders to get to the team members. To hide the complexity of navigating through these layers, we can create model defined functions that allow easier and more direct access to these navigation properties.

To create and use these functions, do the following:

1. Right-click the .edmx file in the Solution Explorer and click Open With ➤ XML Editor. This will open the .edmx file in the XML Editor.
2. Insert the code in Listing 11-7 just below the **<Schema>** tag in the conceptual models section of the .edmx file. This defines the functions in the model.
3. Insert into and query the model using code similar to pattern shown in Listing 11-8.

Listing 11-7. Model defined functions for navigating the associate hierarchy

```
<Function Name="GetProjectManager" ReturnType="EFRecipesModel.ProjectManager">
  <Parameter Name="teammember" Type="EFRecipesModel.TeamMember" />
  <DefiningExpression>
    treat(teammember.Manager.Manager as EFRecipesModel.ProjectManager)
  </DefiningExpression>
</Function>

<Function Name="GetSupervisor" ReturnType="EFRecipesModel.Supervisor">
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
  <Parameter Name="teammember" Type="EFRecipesModel.TeamMember" />
  <DefiningExpression>
    treat(EFRecipesModel.GetProjectManager(teammember).Manager as
       EFRecipesModel.Supervisor)
  </DefiningExpression>
</Function>
```

Listing 11-8. Using both eSQL and LINQ to query the model

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var john = new Supervisor { Name = "John Smith" };
            var steve = new Supervisor {Name = "Steve Johnson"};
            var jill = new ProjectManager { Name = "Jill Masterson",
                                            Manager = john };
            var karen = new ProjectManager { Name = "Karen Carns",
                                             Manager = steve };
            var bob = new TeamLead { Name = "Bob Richardson", Manager = karen };
            var tom = new TeamLead { Name = "Tom Landers", Manager = jill };
            var nancy = new TeamMember { Name = "Nancy Jones", Manager = tom };
            var stacy = new TeamMember { Name = "Stacy Rutgers",
                                         Manager = bob };
            context.Associates.AddObject(john);
            context.Associates.AddObject(steve);
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Using eSQL...");
            var emps = context.Associates.OfType<TeamMember>()
                .Where(@"EFRecipesModel.GetProjectManager(it).Name =
                       @projectManager ||
                       EFRecipesModel.GetSupervisor(it).Name == @supervisor",
                new ObjectParameter("projectManager", "Jill Masterson"),
                new ObjectParameter("supervisor", "Steve Johnson"));
            Console.WriteLine("Team members that report up to either");
            Console.WriteLine("Project Manager Jill Masterson ");
            Console.WriteLine("or Supervisor Steve Johnson");
            foreach (var emp in emps)
            {
                Console.WriteLine("\tAssociate: {0}", emp.Name);
            }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine();
            Console.WriteLine("Using LINQ...");
            var emps = from e in context.Associates.OfType<TeamMember>()
                       where MyFunctions.GetProjectManager(e).Name ==
                        "Jill Masterson" ||
                       MyFunctions.GetSupervisor(e).Name == "Steve Johnson"
                       select e;
            Console.WriteLine("Team members that report up to either");
            Console.WriteLine("Project Manager Jill Masterson ");
            Console.WriteLine("or Supervisor Steve Johnson");
            foreach (var emp in emps)
            {
                Console.WriteLine("\tAssociate: {0}", emp.Name);
            }
        }
    }
}

public class MyFunctions
{
    [EdmFunction("EFRecipesModel", "GetProjectManager")]
    public static ProjectManager GetProjectManager(TeamMember member)
    {
        throw new NotSupportedException("Direct calls not supported.");
    }

    [EdmFunction("EFRecipesModel", "GetSupervisor")]
    public static Supervisor GetSupervisor(TeamMember member)
    {
        throw new NotSupportedException("Direct calls not supported.");
    }
}
```

The output of the code in Listing 11-8 is the following:

```
Using eSQL...

Team members that report up to either

Project Manager Jill Masterson

or Supervisor Steve Johnson

        Associate: Nancy Jones

        Associate: Stacy Rutgers
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Using LINQ...

Team members that report up to either

Project Manager Jill Masterson

or Supervisor Steve Johnson

        Associate: Nancy Jones

        Associate: Stacy Rutgers
```

How It Works

In the **GetSupervisor()** function in Listing 11-7, we need to make three hops through the Manager navigation property. The first one gets the team lead from the team member, the second one gets the project manager from the team lead, and the final one gets the supervisor from the project manager. We already created the **GetProjectManager()** function earlier in Listing 11-7, so we can leverage that function to simplify the implementation of the **GetSupervisor()** function.

We use the **treat()** eSQL operator to cast an instance of Associate to its concrete type, which is either ProjectManager or Supervisor. If we didn't use the **treat()** operator, Entity Framework would raise an exception complaining that it cannot map the instance of Associate to ProjectManager or Supervisor.

In Listing 11-8, using the **GetProjectManager()** and **GetSupervisor()** functions allows us to simplify the code by hiding all the traversal through the object graph via the Manager navigation property.

Because we are not returning **IQueryable<T>** from our model defined function, we didn't need to provide an implementation of the stubs we require to use these functions in the LINQ query.

11-5. Returning an Anonymous Type From a Model Defined Function

Problem

You want to create a model defined function that returns an anonymous type.

Solution

Let's say you have a model for hotel reservations like the one shown in Figure 11-5.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

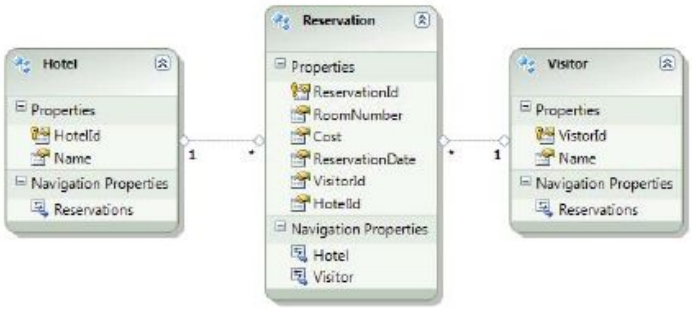


Figure 11-5. *A model for hotel reservations*

You want to retrieve the total number of reservations and the total room revenue for each visitor. Because you will need this information in several places, you want to create a model defined function that takes in a search parameter and returns a collection of anonymous types containing the summary information for each visitor.

To create and use this model defined function, do the following:

1. Right-click the .edmx file in the Solution Explorer and click Open With ➤ XML Editor. This will open the .edmx file in the XML Editor.
2. Insert the code in Listing 11-9 just below the **<Schema>** tag in the conceptual models section of the .edmx file. This defines the function in the model.
3. Insert into and query the model using code similar to pattern shown in Listing 11-10.

Listing 11-9. The VisitorSummary() model defined function

```
<Function Name="VisitorSummary">
  <Parameter Name="StartDate" Type="Edm.DateTime" />
  <Parameter Name="Days" Type="Edm.Int32" />
  <ReturnType>
    <CollectionType>
      <RowType>
        <Property Name="Name" Type="Edm.String" />
        <Property Name="TotalReservations" Type="Edm.Int32" />
        <Property Name="BusinessEarned" Type="Edm.Decimal" />
      </RowType>
    </CollectionType>
  </ReturnType>
  <DefiningExpression>
    Select
    r.Visitor.Name,
    COUNT(r.ReservationId) as TotalReservations,
    SUM(r.Cost) as BusinessEarned
    from EFRecipesEntities.Reservations as r
    where r.ReservationDate between StartDate and
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    AddDays(StartDate,Days)
    group by r.Visitor.Name
  </DefiningExpression>
</Function>
```

Listing 11-10. Querying the model using the VistorySummary() model defined function

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var hotel = new Hotel { Name = "Five Seasons Resort" };
            var v1 = new Visitor { Name = "Alex Stevens" };
            var v2 = new Visitor { Name = "Joan Hills" };
            var r1 = new Reservation { Cost = 79.99M, Hotel = hotel,
                ReservationDate = DateTime.Parse("2/19/2010"), Visitor = v1 };
            var r2 = new Reservation { Cost = 99.99M, Hotel = hotel,
                ReservationDate = DateTime.Parse("2/17/2010"), Visitor = v2 };
            var r3 = new Reservation { Cost = 109.99M, Hotel = hotel,
                ReservationDate = DateTime.Parse("2/18/2010"), Visitor = v1 };
            var r4 = new Reservation { Cost = 89.99M, Hotel = hotel,
                ReservationDate = DateTime.Parse("2/17/2010"), Visitor = v2 };
            context.Hotels.AddObject(hotel);
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Using eSQL...");
            var esql = @"Select value v from
             EFRecipesModel.VisitorSummary(DATETIME'2010-02-16 00:00', 7) as v";
            var visitors = context.CreateQuery<DbDataRecord>(esql);
            foreach (var visitor in visitors)
            {
                Console.WriteLine("{0}, Total Reservations: {1}, Revenue: {2:C}",
                    visitor["Name"], visitor["TotalReservations"],
                    visitor["BusinessEarned"]);
            }
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine();
            Console.WriteLine("Using LINQ...");
            var visitors = from v in
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                        context.VisitorSummary(DateTime.Parse("2/16/2010"), 7)
                        select v;
            foreach (var visitor in visitors)
            {
                Console.WriteLine("{0}, Total Reservations: {1}, Revenue: {2:C}",
                    visitor["Name"], visitor["TotalReservations"],
                    visitor["BusinessEarned"]);
            }
        }
    }
}

partial class EFRecipesEntities
{
    [EdmFunction("EFRecipesModel", "VisitorSummary")]
    public IQueryable<DbDataRecord> VisitorSummary(DateTime StartDate, int Days)
    {
        return this.QueryProvider.CreateQuery<DbDataRecord>(
            Expression.Call(
            Expression.Constant(this),
            (MethodInfo)MethodInfo.GetCurrentMethod(),
            new Expression[] { Expression.Constant(StartDate),
                               Expression.Constant(Days) }
            ));
    }
}
```

The output from the code in Listing 11-10 is the following:

```
Using eSQL...

Alex Stevens, Total Reservations: 2, Revenue: $189.98

Joan Hills, Total Reservations: 2, Revenue: $189.98



Using LINQ...

Alex Stevens, Total Reservations: 2, Revenue: $189.98

Joan Hills, Total Reservations: 2, Revenue: $189.98
```

How It Works

In Listing 11-9, for the definition of the **VisitorSummary()** function, we group the results by visitor, which is the navigation property exposed on the entity. To get the total count of reservations for each visitor, we use the eSQL **Count()** function. To get the total revenue, we use the **Sum()** function.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In the function, we shape the results as a collection of rows of three values: Name, TotalReservations, and BusinessEarned. Here we use the `<CollectionType>` and `<RowType>` tags to indicate the return type. In CLR terms, this is a collection of DbDataRecords.

To use the function in a LINQ query, we create a CLR method that returns `IQueryable<DbDataRecord>`. As in the previous recipes, we decorated the method with the `EdmFunction()` attribute. However, because we are returning an `IQueryable<T>`, we need to implement the body of the method to include the function call in the expression tree. And, because we need access to the QueryProvider in our ObjectContext to return an `IQueryable<T>`, we need to implement this method inside the EFRecipesEntities class.

11-6. Returning a Complex Type From a Model Defined Function

Problem

You want to return a complex type from a model defined function.

Solution

Suppose we have a model for patients and their visits to a local hospital. This model is shown in Figure 11-6.

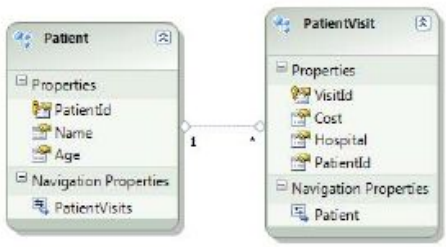


Figure 11-6. *A model for patient visits*

You want to create a model defined function that returns summary information about the patient with their name, the total number of visits, and their accumulated bill. Additionally, you want to filter the results to include only patients over 40 years old.

To create and use the model defined function, do the following:

1. Right-click the designer and select Add ➤ Complex Type.
2. Right-click the new complex type in the Model Browser. Rename the type to **VisitSummary** and add the following properties:
    a. Name of type String, not nullable

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

b. TotalVisits of type Int32, not nullable

c. TotalCost of type Decimal, not nullable

3. Right-click the .edmx file in the Solution Explorer and click Open With ➤ XML Editor. This will open the .edmx file in the XML Editor.

4. Insert the code in Listing 11-11 just below the **<Schema>** tag in the conceptual models section of the .edmx file. This defines the function in the model.

5. Insert into and query the model using code similar to pattern shown in Listing 11-12.

Listing 11-11. The GetVisitSummary() model defined function

```
<Function Name="GetVisitSummary"  ReturnType="Collection(EFRecipesModel.VisitSummary)">
  <DefiningExpression>
    select VALUE EFRecipesModel.VisitSummary(pv.Patient.Name,
                 Count(pv.VisitId),Sum(pv.Cost))
    from EFRecipesEntities.PatientVisits as pv
    group by pv.Patient.Name
  </DefiningExpression>
</Function>
```

Licensed by Matt Allen 2175280

Listing 11-12. Using eSQL and LINQ with the VisitSummary() function to query the model

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            string hospital = "Oakland General";
            var p1 = new Patient { Name = "Robin Rosen", Age = 41 };
            var p2 = new Patient { Name = "Alex Jones", Age = 39 };
            var p3 = new Patient { Name = "Susan Kirby", Age = 54 };
            var v1 = new PatientVisit { Cost = 98.38M, Hospital = hospital,
                                        Patient = p1 };
            var v2 = new PatientVisit { Cost = 1122.98M, Hospital = hospital,
                                        Patient = p1 };
            var v3 = new PatientVisit { Cost = 2292.72M, Hospital = hospital,
                                        Patient = p2 };
            var v4 = new PatientVisit { Cost = 1145.73M, Hospital = hospital,
                                        Patient = p3 };
            var v5 = new PatientVisit { Cost = 2891.07M, Hospital = hospital,
                                        Patient = p3 };
            context.Patients.AddObject(p1);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            context.Patients.AddObject(p2);
            context.Patients.AddObject(p3);
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Query using eSQL...");
            var esql = @"Select value ps from EFRecipesEntities.Patients
                         as p join EFRecipesModel.GetVisitSummary()
                         as ps on p.Name = ps.Name where p.Age > 40";
            var patients = context.CreateQuery<VisitSummary>(esql);
            foreach (var patient in patients)
            {
                Console.WriteLine("{0}, Visits: {1}, Total Bill: {2}",
                    patient.Name, patient.TotalVisits.ToString(),
                    patient.TotalCost.ToString("C"));
            }
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine();
            Console.WriteLine("Query using LINQ...");
            var patients = from p in context.Patients
                           join ps in context.GetVisitSummary() on p.Name equals
                            ps.Name
                           where p.Age >= 40
                           select ps;
            foreach (var patient in patients)
            {
                Console.WriteLine("{0}, Visits: {1}, Total Bill: {2}",
                    patient.Name, patient.TotalVisits.ToString(),
                    patient.TotalCost.ToString("C"));
            }
        }
    }
}

partial class EFRecipesEntities
{
    [EdmFunction("EFRecipesModel", "GetVisitSummary")]
    public IQueryable<VisitSummary> GetVisitSummary()
    {
        return this.QueryProvider.CreateQuery<VisitSummary>(
            Expression.Call(Expression.Constant(this),
              (MethodInfo)MethodInfo.GetCurrentMethod()));
    }
}
```

The code in Listing 11-12 produces the following output:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Query using eSQL...

Robin Rosen, Visits: 2, Total Bill: $1,221.36

Susan Kirby, Visits: 2, Total Bill: $4,036.80


Query using LINQ...

Robin Rosen, Visits: 2, Total Bill: $1,221.36

Susan Kirby, Visits: 2, Total Bill: $4,036.80
```

How It Works

We started by creating the complex type in the model. With the complex type created, we defined the **GetVisitSummary()** function in Listing 11-11 as returning a collection of our newly created complex type. Notice that the constructor for our complex type takes in parameters in the same order as those defined by our complex type. You might need to double-check in the .edmx file to make sure that the designer created the complex type properties in the order in which you created them interactively.

Because our function returns **IQueryable<VisitSummary>**, we need to implement the bootstrapping code. And, because we need to get access to the QueryProvider inside our ObjectContext, we need to implement the method in a partial class of our EFRecipesEntities class, which is our ObjectContext.

You might be wondering when you would return a collection of complex types rather than a collection of anonymous types from a function. If you used the function in a LINQ query, the bootstrapping method would need to return **IQueryable<DbDataRecord>** for the anonymous type. However, although this collection could not be further filtered, a collection of complex types could be further filtered.

11-7. Returning a Collection of Entity References From a Model Defined Function

Problem

You want to return a collection of entity references from a model defined function.

Solution

Let's say you have a model, such as the one in Figure 11-7, for events and their sponsors. Sponsors provide different levels of financial support for events. Platinum sponsors provide the highest level of financial support.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 11-7. *A model for events and their sponsors*

You want to create a model defined function that returns a collection of all the sponsors who are at the Platinum level. Because you need only the entity key information for the sponsor, the function needs to return only a collection of references to the sponsors.

To create and use the model defined function, do the following:

1. Right-click the .edmx file in the Solution Explorer and click Open With ➤ XML Editor. This will open the .edmx file in the XML Editor.
2. Insert the code in Listing 11-13 just below the `<Schema>` tag in the conceptual models section of the .edmx file. This defines the function in the model.
3. Insert into and query the model using code similar to pattern shown in Listing 11-14.

Listing 11-13. The definition of the PlatinumSponsors() function

```
<Function Name="PlatinumSponsors">
    <ReturnType>
        <CollectionType>
          <ReferenceType Type="EFRecipesModel.Sponsor" />
        </CollectionType>
    </ReturnType>
    <DefiningExpression>
       select value ref(s)
          from EFRecipesEntities.Sponsors as s
          where s.SponsorType.Description == 'Platinum'
    </DefiningExpression>
</Function>
```

Listing 11-14. Using eSQL and our PlatinumSponsors() function to find all events with Platinum level sponsors

```
using (var context = new EFRecipesEntities())
{
    var platst = new SponsorType { Description = "Platinum" };
    var goldst = new SponsorType { Description = "Gold" };
    var sp1 = new Sponsor { Name = "Rex's Auto Body Shop",
                            SponsorType = goldst };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    var sp2 = new Sponsor { Name = "Midtown Eye Care Center",
                            SponsorType = platst };
    var sp3 = new Sponsor { Name = "Tri-Cities Ford",
                            SponsorType = platst };
    var ev1 = new Event { Name = "OctoberFest", Sponsor = sp1 };
    var ev2 = new Event { Name = "Concerts in the Park", Sponsor = sp2 };
    var ev3 = new Event { Name = "11th Street Art Festival", Sponsor = sp3 };
    context.Events.AddObject(ev1);
    context.Events.AddObject(ev2);
    context.Events.AddObject(ev3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Events with Platinum Sponsors");
    Console.WriteLine("============================");
    var esql = @"select value e from EFRecipesEntities.Events as e where
                ref(e.Sponsor) in (EFRecipesModel.PlatinumSponsors())";
    var events = context.CreateQuery<Event>(esql);
    foreach (var ev in events)
    {
        Console.WriteLine(ev.Name);
    }
}
```

The output of the code in Listing 11-13 is the following:

```
Events with Platinum Sponsors

============================

Concerts in the Park

11th Street Art Festival
```

How It Works

The **<ReferenceType>** element in the conceptual model denotes a reference to an entity type. This means that we are returning a reference to an entity, not the complete entity. Our model defined function returns a collection of references to Platinum level sponsors. To illustrate using our function, we created an eSQL expression in Listing 11-13 to get all the events with Platinum level sponsors. There are, of course, lots of different ways to get the events sponsored by Platinum level sponsors, but by encapsulating the collection of Platinum level sponsors in our model defined function, we introduce a bit of code reusability.

We didn't show a corresponding use in a LINQ query because the bootstrapping code would need to return an **IQueryable<EntityKey>**, which is fine, but a subsequent **Contains** clause would not work because the result is not strongly typed.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

11-8. Using Canonical Functions in eSQL

Problem

You want to call a Canonical Function in your eSQL query. A *canonical function* is an eSQL function that is natively supported by all data providers. Examples include **Sum()**, **Count()**, and **Avg()**.

Solution

Suppose we have a model for customers and their orders, as shown in Figure 11-8.

Figure 11-8. *A model for customers and their orders*

You want to retrieve the number of orders and the total purchase amount made by customers who have placed orders above the average order.

To create and use this query, follow the pattern shown in Listing 11-14.

Listing 11-14. Querying the model in eSQL using the Sum(), Count(), and Avg() functions

```
using (var context = new EFRecipesEntities())
{
    var c1 = new Customer { Name = "Jill Masters", City = "Raytown" };
    var c2 = new Customer { Name = "Bob Meyers", City = "Austin" };
    var c3 = new Customer { Name = "Robin Rosen", City = "Dallas" };
    var o1 = new Order { OrderAmount = 12.99M, Customer = c1 };
    var o2 = new Order { OrderAmount = 99.39M, Customer = c2 };
    var o3 = new Order { OrderAmount = 101.29M, Customer = c3 };
    context.Orders.AddObject(o1);
    context.Orders.AddObject(o2);
    context.Orders.AddObject(o3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Customers with above average total purchases");
    var esql = @"select o.Customer.Name, count(o.OrderId) as TotalOrders,
                 Sum(o.OrderAmount) as TotalPurchases
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                from EFRecipesEntities.Orders as o
                where o.OrderAmount >
                  anyelement(select value Avg(o.OrderAmount) from
                             EFRecipesEntities.Orders as o)
                group by o.Customer.Name";
    var summary = context.CreateQuery<DbDataRecord>(esql);
    foreach (var item in summary)
    {
        Console.WriteLine("\t{0}, Total Orders: {1}, Total: {2:C}",
            item["Name"], item["TotalOrders"], item["TotalPurchases"]);
    }
}
```

The output of the code in Listing 11-14 is the following:

```
Customers with above average total purchases

        Bob Meyers, Total Orders: 1, Total: $99.39

        Robin Rosen, Total Orders: 1, Total: $101.29
```

How It Works

In this recipe, we used the canonical functions **Count()**, **Sum()**, and **Avg()**. These functions are independent of the data store, which means that they are portable and return types in the EDM space rather than data store-specific types or CLR types.

The current release of Entity Framework introduced the EntityFunctions class, which exposes these canonical functions to LINQ queries as well.

11-9. Using Canonical Functions in LINQ

Problem

You want to use canonical functions in a LINQ query.

Solution

Let's say you have a model for movie rentals like the one in Figure 11-9. The MovieRental entity holds the date the movie was rented and the date it was returned, as well as any late fees that have been accumulated.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

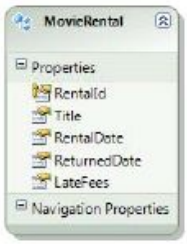


Figure 11-9. *The MovieRental entity that has the dates for a rental period along with any late fees*

You want to retrieve all the movies that were returned more than 10 days after they were rented. These are the late movies.

To create and use this query, follow the pattern shown in Listing 11-15.

Listing 11-15. Retrieving the late movies using the DateDiff() function

```
using (var context = new EFRecipesEntities())
{
    var mr1 = new MovieRental { Title = "A Day in the Life",
                RentalDate = DateTime.Parse("2/19/2010"),
                ReturnedDate = DateTime.Parse("3/4/2010"),LateFees = 3M };
    var mr2 = new MovieRental { Title = "The Shortest Yard",
                RentalDate = DateTime.Parse("3/15/2010"),
                ReturnedDate = DateTime.Parse("3/20/2010"), LateFees = 0M };
    var mr3 = new MovieRental { Title = "Jim's Story",
                RentalDate = DateTime.Parse("3/2/2010"),
                ReturnedDate = DateTime.Parse("3/19/2010"), LateFees = 3M };
    context.MovieRentals.AddObject(mr1);
    context.MovieRentals.AddObject(mr2);
    context.MovieRentals.AddObject(mr3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Movie rentals late returns");
    Console.WriteLine("=========================");
    var late = from r in context.MovieRentals
               where EntityFunctions.DiffDays(r.RentalDate, r.ReturnedDate) > 10
               select r;
    foreach (var rental in late)
    {
        Console.WriteLine("{0} was {1} days late, fee: {2}", rental.Title,
                          (rental.ReturnedDate - rental.RentalDate).Days - 10,
                          rental.LateFees.ToString("C"));
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The output of the code in Listing 11-15 is the following:

```
Movie rentals late returns

=========================

A Day in the Life was 3 days late, fee: $3.00

Jim's Story was 7 days late, fee: $3.00
```

How It Works

Canonical functions, which are defined in Entity Framework, are data source-agnostic and supported by all data providers. The types returned from canonical functions are defined in terms of types from the Entity Data Model.

In this recipe, we used the DiffDays() function to calculate the number of days between the start and end of the rental period. Because **DiffDays()** is a canonical function, it will be implemented by all providers.

Best Practice

When should I use EntityFunctions? Entity Framework provides translations for some expressions into the canonical functions, but the translation is limited. Not every CLR method will translate to the corresponding canonical function.

Here's the best practice. If there is a translation available, use it. It makes the code easier to read. If there is no translation available, use the EntityFunction class to explicitly call the canonical function, as in the following:

```
var laterentals = from r in context.MovieRentals
                  where (r.ReturnedDate - r.RentalDate).Days > 10
                  select r;
```

does not translate to the Canonical Function, so you should use,

```
var laterentals = from r in context.MovieRentals
                  where EntityFunctions.DiffDays(r.RentalDate,
                                                 r.ReturnedDate) > 10
                  select r;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

11-10. Calling Database Functions in eSQL

Problem

You want to call a database function in an eSQL statement.

Solution

Let's say you have an eCommerce website and you need to find all the customers within a certain distance of a given ZIP code. Your model might look like the one in Figure 11-10.

Figure 11-10. *WebCustomer and Zip entities in a model*

We'll need to pull out some basic math functions to get this to work. Unfortunately, Entity Framework does not have the canonical functions we need, so we'll have to use the functions available in the data store.

Use the pattern in Listing 11-16 to call the database functions from an eSQL expression.

Listing 11-16. Using database functions to determine the distance between a customer and a given ZIP code

```
using (var context = new EFRecipesEntities())
{
    var c1 = new WebCustomer { Name = "Alex Stevens", Zip = "76039" };
    var c2 = new WebCustomer { Name = "Janis Jones", Zip = "76040" };
    var c3 = new WebCustomer { Name = "Cathy Robins", Zip = "76111" };
    context.Zips.AddObject(new Zip { Latitude = 32.834298M,
                                     Longitude = -32.834298M,
                                     ZipCode = "76039" });
    context.Zips.AddObject(new Zip { Latitude = 32.835298M,
                                     Longitude = -32.834798M,
                                     ZipCode = "76040" });
    context.Zips.AddObject(new Zip { Latitude = 33.834298M,
                                     Longitude = -31.834298M,
                                     ZipCode = "76111" });
    context.WebCustomers.AddObject(c1);
    context.WebCustomers.AddObject(c2);
    context.WebCustomers.AddObject(c3);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    string esql = @"select value c
                    from EFRecipesEntities.WebCustomers as c
                    join
                    (SELECT z.ZipCode,
                      3958.75 * (SqlServer.Atan(SqlServer.Sqrt(1 -
                       SqlServer.power(((SqlServer.Sin(t2.Latitude/57.2958M) *
                           SqlServer.Sin(z.Latitude/57.2958M)) +
                           (SqlServer.Cos(t2.Latitude/57.2958M) *
                           SqlServer.Cos(z.Latitude/57.2958M) *
                            SqlServer.Cos((z.Longitude/57.2958M) -
                           (t2.Longitude/57.2958M)))), 2)) /(
                             ((SqlServer.Sin(t2.Latitude/57.2958M) *
                             SqlServer.Sin(z.Latitude/57.2958M)) +
                              (SqlServer.Cos(t2.Latitude/57.2958M) *
                               SqlServer.Cos(z.Latitude/57.2958M) *
                               SqlServer.Cos((z.Longitude/57.2958M) -
                                  (t2.Longitude/57.2958M))))))
                     ) as DistanceInMiles
                     FROM EFRecipesEntities.Zips AS z join
                      (select top(1) z2.Latitude as Latitude,z2.Longitude as
                       Longitude
                       from EFRecipesEntities.Zips as z2
                       where z2.ZipCode = @Zip
                      ) as t2 on 1 = 1
                    ) as matchingzips on matchingzips.ZipCode = c.Zip
                   where matchingzips.DistanceInMiles <= @RadiusInMiles";

    var custs = context.CreateQuery<WebCustomer>(esql,
                    new ObjectParameter("Zip", "76039"),
                    new ObjectParameter("RadiusInMiles", 5));
    Console.WriteLine("Customers within 5 miles of 76039");
    foreach (var cust in custs)
    {
        Console.WriteLine("Customer: {0}", cust.Name);
    }
}
```

The output of the code in Listing 11-16 is the following:

```
Customers within 5 miles of 76039

Customer: Alex Stevens

Customer: Janis Jones
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Okay, the eSQL is a little complex, but the complexity is because we're calling a bunch of database functions. Using the database functions in eSQL is fairly simple. These functions are available in the SqlServer namespace. Not all database functions are available in eSQL, so check the current Microsoft documentation to get a complete list.

In this example, the Zip entity has the Latitude and Longitude for each ZIP code. These values represent the geographic location of the center of the ZIP code. To calculate the between two ZIP codes involves a bit of math. Luckily, the database side provides the necessary functions to do the calculation.

11-11. Calling Database Functions in LINQ

Problem

You want to call a database function in a LINQ query.

Solution

Let's say you have an Appointment entity in your model and you want to query for all the appointments you have on a given day of the week. The Appointment entity might look like the one in Figure 11-11.

Figure 11-11. *An Appointment entity with the start and end times for appointments*

If we want to find all the appointments for Thursday, we can't use the CLR `enum DayOfWeek.Thursday` to compare with the StartsAt property in a `where` clause because this does not translate to a data store statement. We need to use the pattern shown in Listing 11-17.

Listing 11-17. Using a database function in a LINQ query

```
using (var context = new EFRecipesEntities())
{
    var app1 = new Appointment { StartsAt = DateTime.Parse("4/7/2010 14:00"),
                         GoesTo = DateTime.Parse("4/7/2010 15:00") };
    var app2 = new Appointment { StartsAt = DateTime.Parse("4/8/2010 9:00"),
                         GoesTo = DateTime.Parse("4/8/2010 11:00") };
    var app3 = new Appointment { StartsAt = DateTime.Parse("4/8/2010 13:00"),
                         GoesTo = DateTime.Parse("4/7/2010 15:00") };
    context.Appointments.AddObject(app1);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.Appointments.AddObject(app2);
    context.Appointments.AddObject(app3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var apps = from a in context.Appointments
               where SqlFunctions.DatePart("WEEKDAY", a.StartsAt) == 5
               select a;
    Console.WriteLine("Appointments for Thursday");
    Console.WriteLine("========================");
    foreach (var appointment in apps)
    {
        Console.WriteLine("Appointment from {0} to {1}",
                          appointment.StartsAt.ToShortTimeString(),
                          appointment.GoesTo.ToShortTimeString());
    }
}
```

The output of the code in Listing 11-17 is the following:

```
Appointments for Thursday

========================

Appointment from 9:00 AM to 11:00 AM

Appointment from 1:00 PM to 3:00 PM
```

How It Works

Database functions are available for use in both eSQL and LINQ queries. These functions are exposed via methods in the SqlFunctions class. Because these functions execute on the database side, the behavior you get might differ slightly from what you would expect on the .NET side. For example, `DayOfWeek.Thursday` evaluates to 4 on the .NET side. On the database side, Thursday is the fifth day of the week, so we check for a value of 5.

As with database functions in eSQL, not all database functions are available for LINQ queries. Check the current documentation from Microsoft for a complete list of the available functions.

11-12. Defining Built-in Functions

Problem

You want to define a built-in function for use in an eSQL or LINQ query.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you want to use the IsNull function in the database, but this function is not currently exposed by Entity Framework for either eSQL or LINQ. Suppose we have a WebProduct entity in our model like the one shown in Figure 11-12.

Figure 11-12. *A WebProduct entity in our model*

To expose this database function for your queries, do the following:

1. Right-click the .edmx file in the Solution Explorer and click Open With ➤ XML Editor. This will open the .edmx file in the XML Editor.
2. Insert the code in Listing 11-18 just below the `<Schema>` tag in the storage models section of the .edmx file. This defines the functions in the storage layer.
3. Insert into and query the model using code similar to pattern shown in Listing 11-19.

Listing 11-18. Defining our function in the storage layer

```
<Function Name="ISNULL" ReturnType="varchar" BuiltIn="true" Schema="dbo">
          <Parameter Name="expr1" Type="varchar"  Mode="In" />
          <Parameter Name="expr2" Type="varchar"  Mode="In" />
</Function>
```

Listing 11-19. Using the ISNULL() function in an eSQL and LINQ query

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var w1 = new WebProduct { Name = "Camping Tent",
                   Description = "Family Camping Tent, Color Green" };
            var w2 = new WebProduct { Name = "Chemical Light" };
            var w3 = new WebProduct { Name = "Ground Cover",
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                Description = "Blue ground cover" };
            context.WebProducts.AddObject(w1);
            context.WebProducts.AddObject(w2);
            context.WebProducts.AddObject(w3);
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Query using eSQL...");
            var esql = @"select value
                 EFRecipesModel.Store.ISNULL(p.Description,p.Name)
                 from EFRecipesEntities.WebProducts as p";
            var prods = context.CreateQuery<string>(esql);
            foreach (var prod in prods)
            {
                Console.WriteLine("Product Description: {0}", prod);
            }
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine();
            Console.WriteLine("Query using LINQ...");
            var prods = from p in context.WebProducts
                        select BuiltinFunctions.ISNULL(p.Description, p.Name);
            foreach (var prod in prods)
            {
                Console.WriteLine(prod);
            }
        }
    }
}

public class BuiltinFunctions
{
    [EdmFunction("EFRecipesModel.Store", "ISNULL")]
    public static string ISNULL(string check_expression, string replacementvalue)
    {
        throw new NotSupportedException("Direct calls are not supported.");
    }
}
```

The output from the code in Listing 11-19 is the following:

```
Query using eSQL...

Product Description: Family Camping Tent, Color Green

Product Description: Chemical Light
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Product Description: Blue ground cover


Query using LINQ...

Family Camping Tent, Color Green

Chemical Light

Blue ground cover
```

How It Works

In the definition of the **ISNULL()** function in Listing 11-17, we need to match the name of the database function with our function's name. Both have to be the same in spelling but not in case.

We defined the function not in the conceptual layer, as in previous recipes in this chapter, but in the store layer. This function is already available in the database; we are simply surfacing it in the store layer for our use.

When we use the function in the eSQL statement, we need to fully qualify the namespace for the function. Here that fully qualified name is **EFRecipesModel.Store.ISNULL()**.

To use the function in a LINQ query, we need to create the bootstrapping method. We are not returning an **IQueryable<T>**, so no implementation of the method is required.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 12

Customizing Entity Framework Objects

The recipes in this chapter explore some of the customizations that can be applied to objects and to the processes in Entity Framework. These recipes cover many of the "behind the scenes" things you can do to make your code more uniform by pushing concerns about things like business rule enforcement from the details of your application to a central, application-wide implementation.

We start off this chapter with a recipe that shows you how to have your own code execute anytime `SaveChanges()` is called in your application. This recipe and a few others are particularly useful if you want to enforce business rules from a single spot in your application.

In other recipes, we show you how to track database connections, how to automate responses to collection changes, how to implement cascading deletes, how to assign default values, and how to work with strongly typed XML properties.

The common thread of all these recipes is extending the objects and processes in Entity Framework to make your code more resilient, uniform, and maintainable.

12-1. Executing Code When SaveChanges() Is Called

Problem

You want to execute code anytime `SaveChanges()` is called in a data context.

Solution

Let's say you have a model that represents a job applicant. As part of the model, you want the file containing the applicant's resume to be deleted when the applicant's record is deleted. You could find every place in your application where you delete an applicant's record, but you want a more consistent and unified approach.

Your model looks like the one in Figure 12-1.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 12-1. *A model for job applicant*

To ensure that the applicant's resume file is deleted when the applicant is deleted, we override the **SavingChanges()** method in the object context. In our overridden method, we need to scan the object context for changes that include deleting instances of the Applicant entity. Next, we need to tell Entity Framework to save the changes by calling the real **SaveChanges()** method. Finally, for each of the deleted Applicants, we need to delete the associated resume file. The code in Listing 12-1 demonstrates this approach.

Listing 12-1. Overriding SaveChanges() to delete the resume file when the applicant is deleted

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var path1 = "AlexJones.txt";
            File.AppendAllText(path1, "Alex Jones\nResume\n...");
            var path2 = "JanisRogers.txt";
            File.AppendAllText(path2, "Janis Rodgers\nResume\n...");
            var app1 = new Applicant { Name = "Alex Jones",
                                       ResumePath = path1 };
            var app2 = new Applicant { Name = "Janis Rogers",
                                       ResumePath = path2 };
            context.Applicants.AddObject(app1);
            context.Applicants.AddObject(app2);
            context.SaveChanges();

            // delete Alex Jones
            context.Applicants.DeleteObject(app1);
            context.SaveChanges();
        }
    }
}

public partial class EFRecipesEntities
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
{
    public override int SaveChanges(SaveOptions options)
    {
        Console.WriteLine("Saving Changes...");
        var applicants = this.ObjectStateManager
                           .GetObjectStateEntries(EntityState.Deleted)
                           .Select(e => e.Entity)
                           .OfType<Applicant>().ToList();
        int changes = base.SaveChanges(options);
        Console.WriteLine("\n{0} applicants deleted",
                           applicants.Count().ToString());
        foreach (var app in applicants)
        {
            File.Delete(app.ResumePath);
            Console.WriteLine("\n{0}'s resume at {1} deleted",
                              app.Name, app.ResumePath);
        }
        return changes;
    }
}
```

The following is the output from the code in Listing 12-1:

```
Saving Changes...

0 applicants deleted

Saving Changes...



1 applicants deleted



Alex Jones's resume at AlexJones.txt deleted
```

How It Works

The code in Listing 12-1 starts by inserting two applicants, each with the path to a resume file that we also created. The goal here is to delete the resume file in a structured way when the instance of the Applicant entity is deleted. We do this by overriding the **SaveChanges()** method.

In our **SaveChanges()** method, we first gather up all the instances of Applicant that have been marked for deletion. These are the ones that will be deleted from the database when we call the real **SaveChanges()** method. We need to get them before we call **SaveChanges()** because after we call **SaveChanges()**, these instances will be detached from the context and we will no longer be able to use this query to retrieve them. Once we have the instances that will be deleted, we call **SaveChanges()** to do

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

the real work of persisting objects to the database. Once the changes have been successfully committed, we can delete the resume files.

There are a couple of possible variations worth noting. We could wrap both our call to **SaveChanges()** and the file deletions in a transaction to ensure that both either succeed or fail together.

Another variation is to use the **DetectChangesBeforeSave** SaveOption in our call to **SaveChanges()**. This option preserves the tracking information in the object context and would allow us to move the query for deleted objects to somewhere after the call the **SaveChanges()**. To clear the tracking information, we would need to call **AcceptAllChanges()** before we leave the method.

Entity Framework does not expose insert, update, and delete events for each entity. However, much of what we would do in these events can be handled, as we have demonstrated here, by overriding the **SaveChanges()** method.

12-2. Validating Property Changes

Problem

You want to validate a value being assigned to a property.

Solution

Let's say you have a model with a User entity. The User entity has properties for the full name and user name for the user. You have a business rule that says each user must have a UserName greater than five characters long. You want to enforce this business rule with code that sets the IsActive property to **false** if the UserName is set to a string less than or equal to five characters; otherwise the IsActive flag is set to **true**. The model is shown in Figure 12-2.

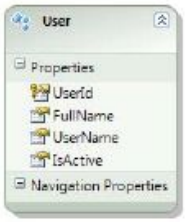


Figure 12-2. *The User entity in our model*

To enforce our business rule, we need to implement the partial methods **OnUserNameChanging()** and **OnUserNameChanged()**. These methods are called during the property change activity and after the property has been changed. The code in Listing 12-2 demonstrates one solution.

Listing 12-2. Monitoring the changing of the UserName property

```
class Program
{
    static void Main(string[] args)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var user1 = new User { FullName = "Robert Meyers",
                                   UserName = "RM" };
            var user2 = new User { FullName = "Karen Kelley",
                                   UserName = "KKelley" };
            context.Users.AddObject(user1);
            context.Users.AddObject(user2);
            context.SaveChanges();
            Console.WriteLine("Users saved to database");
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine();
            Console.WriteLine("Reading users from database");
            foreach (var user in context.Users)
            {
                Console.WriteLine("{0} is {1}, UserName is {2}", user.FullName,
                        user.IsActive ? "Active" : "Inactive", user.UserName);
            }
        }
    }
}

public partial class User
{
    partial void OnUserNameChanging(string value)
    {
        if (value.Length > 5)
            Console.WriteLine("{0}'s UserName changing to {1}, OK!",
                              this.FullName, value);
        else
            Console.WriteLine("{0}'s UserName changing to {1}, Too Short!",
                              this.FullName, value);
    }

    partial void OnUserNameChanged()
    {
        this.IsActive = (this.UserName.Length > 5);
    }
}
```

The following is the output of the code in Listing 12-2:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Robert Meyers's UserName changing to RM, Too Short!
Karen Kelley's UserName changing to KKelley, OK!
Users saved to database

Reading users from database
Robert Meyers's UserName changing to RM, Too Short!
Robert Meyers is Inactive, UserName is RM
Karen Kelley's UserName changing to KKelley, OK!
Karen Kelley is Active, UserName is KKelley
```

How It Works

In the solution, we implement the partial methods **OnUserNameChanging()** and **OnUserNameChanged()** to monitor the property change activity. The **OnUserNameChanging()** method is called when the property value is being set. Here we have an opportunity to throw an exception or, as in our example, simply report that the UserName is being set to a value of five characters or fewer.

The **OnUserNameChanged()** method is called after the property has been changed. Here we simply set the IsActive property based on the length of the final UserName property value.

These partial methods are created by Entity Framework as part of the code generation process. The names of the partial methods are derived from the property names. In our case, each method name included the name of the property. These partial methods are called inside the setter for each property.

You may be wondering a bit about the output of code. Notice that the partial methods are called twice in our example. They are called when the property value is set. They are also called when the User instances are materialized from the database. This second call happens, of course, because the materialization process involves setting the property value from the persisted value in the database.

In addition to these two partial methods, Entity Framework exposes two events for monitoring property changes. These events, **PropertyChanging** and **PropertyChanged**, are raised when any property on an Entity is changed. The sender of the event is the instance of the entity and the PropertyEventArgs parameter contains a PropertyName that holds the name of the property that is changing or has changed. Because these events are fired for any property change on the entity, they can be useful in some scenarios, particularly if you have an entity with many properties. They are somewhat less useful in practical terms because they don't readily expose the current and proposed values for the property.

When our UserName property value changes, the sequence is as follows:

1. **OnUserNameChanging()** method is called.
2. **PropertyChanging** event is raised.
3. **PropertyChanged** event is raised.
4. **OnUserNameChanged()** method is called.

The **PropertyChanging** and **PropertyChanged** events are not raised when a navigation property value is changed. The state of an entity changes only when a scalar or complex property changes.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

12-3. Logging Database Connections

Problem

You want to create a log entry each time a connection is opened or closed to the database.

Solution

Entity Framework exposes a **StateChange** event on the connection for an object context. To create a log entry each time a connection is opened or closed, we need to handle this event.

Suppose our model looks like the one in Figure 12-3. In Listing 12-3, we create a few instances of a Donation and save them to the database. The code implements the **OnContextCreated()** partial method to wire in our handler for the **StateChange** event.

Figure 12-3. *The model with the Donation entity*

Listing 12-3.

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            context.Donations.AddObject(new Donation {
                    DonorName = "Robert Byrd", Amount = 350M });
            context.Donations.AddObject(new Donation {
                    DonorName = "Nancy McVoid", Amount = 250M });
            context.Donations.AddObject(new Donation {
                    DonorName = "Kim Kerns", Amount = 750M });
            Console.WriteLine("About to SaveChanges()");
            context.SaveChanges();
        }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        using (var context = new EFRecipesEntities())
        {
            var list = context.Donations.Where(o => o.Amount > 300M);
            Console.WriteLine("Donations over $300");
            foreach (var donor in list)
            {
                Console.WriteLine("{0} gave {1}", donor.DonorName,
                                  donor.Amount.ToString("C"));
            }
        }
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.Connection.StateChange += (s, e) =>
            {
                var conn = ((EntityConnection)s).StoreConnection;
                Console.WriteLine("{0}: Database: {1}, State: {2}, was: {3}",
                    DateTime.Now.ToShortTimeString(), conn.Database,
                    e.CurrentState, e.OriginalState);
            };
    }
}
```

The following is the output from the code in Listing 12-3:

```
1:09 PM: Database: EFRecipes, State: Open, was: Closed

1:09 PM: Database: EFRecipes, State: Closed, was: Open

About to SaveChanges()

1:09 PM: Database: EFRecipes, State: Open, was: Closed

1:09 PM: Database: EFRecipes, State: Closed, was: Open

Donations over $300

1:09 PM: Database: EFRecipes, State: Open, was: Closed

Robert Byrd gave $350.00

Kim Kerns gave $750.00

1:09 PM: Database: EFRecipes, State: Closed, was: Open
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

To wire in the handler for the **StateChange** event, we implement the **OnContextCreated()** partial method. This partial method is called when the context is created.

Our event handler receives two parameters: the sender of the event and a **StateChangeEventArgs**. This second parameter provides access to the CurrentState of the connection and the OriginalState of the connection. We create a log entry indicating both of these states as well as the time of the event and the associated database.

If you are paying particularly close attention to the order of the log entries, you will notice that in the second **using** block, the connection to the database occurs during the execution of the query in the **foreach** loop, not when the query is constructed. This demonstrates the important concept that queries are executed only when explicitly required. In our case, this execution occurs during the iteration.

12-4. Recalculating a Property Value When an Entity Collection Changes

Problem

You want to recalculate a property value on the entity when its entity collection changes.

Solution

Both **EntityCollection** and **EntityReference** derive from **RelatedEnd**. **RelatedEnd** exposes an **AssociationChanged** event. This event is raised when the association is changed or modified. In particular, this event is raised when an element is added to or removed from a collection.

To recalculate a property values, we implement a handler for the **AssociationChanged** event.

Let's say you have a model with a shopping cart and items for the cart. The model is shown in Figure 12-4.

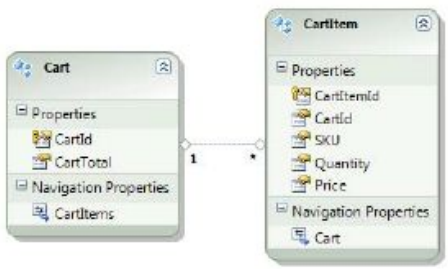


Figure 12-4. *A model for a cart and the cart's items*

The code in Listing 12-4 demonstrates using the **AssociationChanged** event to recalculate the CartTotal property on the Cart entity when items are added to or removed from the CartItems collection.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 12-4. Using the AssociationChanged event to keep the CartTotal in sync with the items in the cart

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var item1 = new CartItem { SKU = "AMM-223", Quantity = 3,
                                       Price = 19.95M };
            var item2 = new CartItem { SKU = "CAMP-12", Quantity = 1,
                                       Price = 59.95M };
            var item3 = new CartItem { SKU = "29292", Quantity = 2,
                                       Price = 4.95M };
            var cart = new Cart { CartTotal = 0 };
            cart.CartItems.Add(item1);
            cart.CartItems.Add(item2);
            cart.CartItems.Add(item3);
            context.Carts.AddObject(cart);
            item1.Quantity = 1;
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            foreach (var cart in context.Carts)
            {
                Console.WriteLine("Cart Total = {0}",
                                  cart.CartTotal.ToString("C"));
                foreach (var item in cart.CartItems)
                {
                    Console.WriteLine("\tSKU = {0}, Qty = {1}, Unit Price = {2}",
                                      item.SKU, item.Quantity.ToString(),
                                      item.Price.ToString("C"));
                }
            }
        }
    }
}

public partial class Cart
{
    public Cart()
    {
        this.CartItems.AssociationChanged += (s, e) =>
            {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                if (e.Action == CollectionChangeAction.Add)
                {
                    var item = e.Element as CartItem;
                    item.PropertyChanged += (ps, pe) =>
                        {
                            if (pe.PropertyName == "Quantity")
                            {
                                this.CartTotal =
                                  this.CartItems.Sum(t => t.Price * t.Quantity);
                                Console.WriteLine("Qty changed, total = {0}",
                                  this.CartTotal.ToString("C"));
                            }
                        };
                }
                this.CartTotal = this.CartItems.Sum(t => t.Price * t.Quantity);
                Console.WriteLine("New total = {0}",
                                  this.CartTotal.ToString("C"));
            };
    }
}
```

The following is the output from the code in Listing 12-4:

```
New total = $59.85

New total = $119.80

New total = $129.70

Qty changed, total = $89.80

Cart Total = $89.80

New total = $89.80

        SKU = AMM-223, Qty = 1, Unit Price = $19.95

        SKU = CAMP-12, Qty = 1, Unit Price = $59.95

        SKU = 29292, Qty = 2, Unit Price = $4.95
```

How It Works

To keep the CartTotal property in sync with the items in the CartItems collection, we need to wire in a handler for the **AssociationChanged** event on the CartItems collection. We do this in the constructor for the Cart entity.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The event handler is a little complicated because we have to consider two cases. In the first case, we're simply adding or removing an item from the cart. Here we just recalculate the total by iterating through the collection and summing the price for each item multiplied by the quantity of the item. To get this sum, we use the **Sum()** method and pass in a lambda expression that multiplies the price and quantity.

In the second case, the entity collection remains the same, but one of the items has its quantity changed. This also affects the cart total and requires that we recalculate. For this case, we wire in a handler for the **PropertyChanged** event whenever we add an item to the cart. This second handler simply recalculates the cart total when the Quantity property changes.

To wire in this second handler, we depend on the Action property exposed in the **CollectionChangedEventArgs**, which is passed as the second parameter to our first event handler. The actions defined are Add, Remove, and Refresh.

Batch operations such as **Load()**, **Clear()**, and **Attach()** raise the **CollectionChangedEvent** just once regardless of how many elements are in the collection. This can be good if your collection contains lots of elements and you are interested, as we are here, in the entire collection. It can, of course, be annoying if you need to track collection changes at a more granular level.

12-5. Automatically Deleting Related Entities

Problem

When an entity is deleted, you want to automatically delete the related entities.

Solution

Suppose you have a table structure that consists of a course, the classes for the course, and the enrollment in each class, as shown in Figure 12-5.

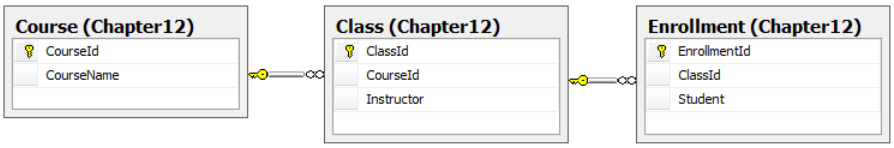


Figure 12-5. *The Course, Class, and Enrollment tables in our database*

Given these tables, you have created a model like the one shown in Figure 12-6.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 12-6. *A model with the Course, Class, and Enrollment entities and their associations*

When a course is deleted from the database, you want all the classes for the course deleted and all the enrollments for the classes deleted as well. To get this to work, we set a cascade delete rule in the database for the relationships. To set this rule, select the relationship in SQL Server Management Studio, view the properties, and select Cascade in the INSERT and UPDATE Specification's Delete Rule.

When these tables are imported into the model, these cascade delete rules will also be imported. You can see this by selecting the one-to-many association between Course and Class and viewing the properties. See Figure 12-7.

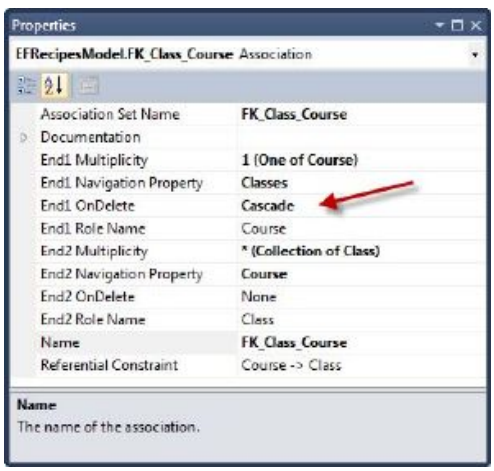


Figure 12-7. *The cascade delete rule from the database was imported into the model and is shown in the properties for the association.*

The cascade delete shown in Figure 12-7 is in the conceptual layer. There is a similar rule present in the store layer. Both these Entity Framework rules and the underlying database cascade delete rule are necessary to keep the object context and the database in sync when objects are deleted.

The code in Listing 12-5 demonstrates the cascade delete.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 12-5. Using the underlying cascade delete rules to delete the related objects

```
using (var context = new EFRecipesEntities())
{
    var course1 = new Course { CourseName = "CS 301" };
    var course2 = new Course { CourseName = "Math 455" };
    var en1 = new Enrollment { Student = "James Folk" };
    var en2 = new Enrollment { Student = "Scott Shores" };
    var en3 = new Enrollment { Student = "Jill Glass" };
    var en4 = new Enrollment { Student = "Robin Rosen" };
    var class1 = new Class { Instructor = "Bill Meyers" };
    var class2 = new Class { Instructor = "Norma Hall" };
    class1.Course = course1;
    class2.Course = course2;
    class1.Enrollments.Add(en1);
    class1.Enrollments.Add(en2);
    class2.Enrollments.Add(en3);
    class2.Enrollments.Add(en4);
    context.Classes.AddObject(class1);
    context.Classes.AddObject(class2);
    context.SaveChanges();
    context.Classes.DeleteObject(class1);
    context.SaveChanges();
}
using (var context = new EFRecipesEntities())
{
    foreach (var course in context.Courses)
    {
        Console.WriteLine("Course: {0}", course.CourseName);
        foreach (var c in course.Classes)
        {
            Console.WriteLine("\tClass: {0}, Instructor: {1}",
                              c.ClassId.ToString(), c.Instructor);
            foreach (var en in c.Enrollments)
            {
                Console.WriteLine("\t\tStudent: {0}", en.Student);
            }
        }
    }
}
```

The following is the output from the code in Listing 12-5:

```
Course: CS 301

Course: Math 455

        Class: 8, Instructor: Norma Hall
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Student: Jill Glass

Student: Robin Rosen
```

How It Works

This recipe has the cascade delete rule both in the database and in the model. In the model, the rule is represented both at the conceptual layer and in the store layer. To keep the object context in sync with the database, we defined the cascade delete in both the database and in the model.

Best Practice

Licensed by Matt Allen 21735280

Now, you may be asking, why do we need this rule in both the model and in the database? Wouldn't it suffice to have the rule either in the database or in the model?

The reason cascade delete exists at the conceptual layer is to keep the objects loaded in the object context in sync with the cascade delete changes made by the database. For example, if we have classes and enrollments for a given course loaded in the object context and we mark the course for deletion, Entity Framework would also mark the course's classes and their enrollments for deletion. All this happens before anything is sent to the database. At the model layer, cascade delete means to mark related entities for deletion. Ultimately, Entity Framework will issue redundant deletes for these entities.

So, if Entity Framework will issue redundant deletes, why not just have the rules in the model and not in the database? Here's why. For Entity Framework to mark entities for deletion, they must be loaded into the object context. Imagine we have a course in the object context, but we haven't loaded the related classes or the related enrollments. If we delete the course, the related classes and enrollments can't be marked for deletion because they are not in the object context. No commands will be sent to the database to delete these related rows. However, if we have the cascade delete rules in place in the database, the database will take care of deleting the rows.

The best practice here is to have the cascade delete rules both in the model and in the database.

If you have added a cascade delete rule to a model, Entity Framework will not overwrite it if you update the model from the database. Unfortunately, if you don't have a cascade delete rule in the model and you update the model from the database and the database has a newly created cascade delete rule, Entity Framework will not add a cascade delete rule in the conceptual layer. You will have to manually add it.

12-6. Deleting All Related Entities

Problem

You want to delete all related entities in the most generic way possible.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

We want to delete all the related entities in a generic way; that is, in a way that will work across all entities without specific reference to any particular entity type. To do this, we will create a method that uses the RelationshipManager to get all the related ends. With these, we can use `CreateSourceQuery()` to retrieve the entities and delete them.

The code in Listing 12-6 demonstrates this method using the model in Figure 12-8. In this model, we have recipes with related ingredients and steps.

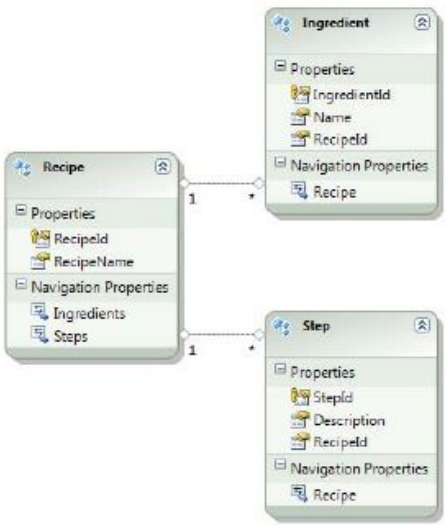


Figure 12-8. *A model with ingredients and steps for each recipe*

Listing 12-6. Demonstrating the DeleteRelatedEntities<>() method

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void DeleteRelatedEntities<T>(T entity, EFRecipesEntities context)
                where T : EntityObject
    {
        var entities = ((IEntityWithRelationships)entity)
                        .RelationshipManager.GetAllRelatedEnds()
                        .SelectMany(e =>
                          e.CreateSourceQuery().OfType<EntityObject>()).ToList();
        foreach (var child in entities)
        {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            context.DeleteObject(child);
        }
        context.SaveChanges();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var recipe1 = new Recipe { RecipeName = "Chicken Risotto" };
            var recipe2 = new Recipe { RecipeName = "Baked Chicken" };
            recipe1.Steps.Add(new Step { Description = "Bring Broth to a boil" });
            recipe1.Steps.Add(new Step { Description =
                                         "Slowly add Broth to Rice" });
            recipe1.Ingredients.Add(new Ingredient { Name = "1 Cup White Rice" });
            recipe1.Ingredients.Add(new Ingredient { Name =
                                         "6 Cups Chicken Broth"});
            recipe2.Steps.Add(new Step { Description =
                                         "Bake at 350 for 35 Minutes" });
            recipe2.Ingredients.Add(new Ingredient { Name = "1 lb Chicken" } );
            context.Recipes.AddObject(recipe1);
            context.Recipes.AddObject(recipe2);
            context.SaveChanges();
            Console.WriteLine("All the Related Entities...");
            ShowRecipes();
            DeleteRelatedEntities(recipe2, context);
            Console.WriteLine("\nAfter Related Entities are Deleted...");
            ShowRecipes();
        }
    }

    static void ShowRecipes()
    {
        using (var context = new EFRecipesEntities())
        {
            foreach (var recipe in context.Recipes)
            {
                Console.WriteLine("\n*** {0} ***", recipe.RecipeName);
                Console.WriteLine("Ingredients");
                foreach (var ingredient in recipe.Ingredients)
                {
                    Console.WriteLine("\t{0}", ingredient.Name);
                }
                Console.WriteLine("Steps");
                foreach (var step in recipe.Steps)
                {
                    Console.WriteLine("\t{0}", step.Description);
                }
            }
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

The following is the output of the code in Listing 12-6:

```
All the Related Entities...


*** Chicken Risotto ***

Ingredients

        1 Cup White Rice

        6 Cups Chicken Broth

Steps

        Bring Broth to a boil

        Slowly add Broth to Rice


*** Baked Chicken ***

Ingredients

        1 lb Chicken

Steps

        Bake at 350 for 35 Minutes


After Related Entities are Deleted...


*** Chicken Risotto ***

Ingredients

        1 Cup White Rice

        6 Cups Chicken Broth
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Steps

        Bring Broth to a boil

        Slowly add Broth to Rice


*** Baked Chicken ***

Ingredients

Steps
```

How It Works

Of course, there is no real performance benefit using the code in Listing 12-6. What is useful about this approach is that it deletes all the related entities without reference to any particular entity type. We could have loaded the second recipe and simply marked each of the ingredients and steps for deletion, but this code snippet would be specific to these entities in this model. The method in Listing 12-6 will work across all entity types and delete all related entities.

12-7. Assigning Default Values

Problem

You want to assign default values to the properties of an entity before it is saved to the database.

Solution

Let's say you have a table similar to the one in Figure 12-9 that holds information about a purchase order. The key, PurchaseOrderId, is a GUID, and there are two columns holding the date and time for the creation and last modification of the object. There is also a comments column that is no longer used and should always be set to "N/A". Because we no longer use the comments, we don't have this property available on the entity. You want to initialize the PurchaseOrderId column, the date fields, the Paid column, and the comment column to default values. Our model is shown in Figure 12-10.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

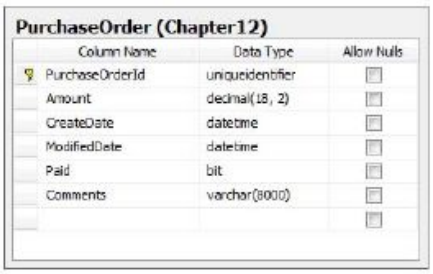


Figure 12-9. *The PurchaseOrder table with several columns that need default values*

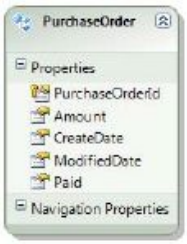


Figure 12-10. *The model created from the PurchaseOrder table in Figure 12-9*

We will illustrate three different ways to set default values. Default values that don't need to be dynamically calculated can be set as the Default Value for the property in the conceptual model. Select the Paid property and view its Properties. Set the Default Value to `false`.

For properties that need to be calculated at runtime, we need to handle the `SavingChanges` event. This is illustrated in Listing 12-7. In this handler, if the object is in the Added state, we set the PurchaseOrderId to a new GUID and set the CreateDate and ModifiedDate fields.

To illustrate setting the default value outside of the conceptual model, we can modify the store layer to set a default value for the comments column. This approach would be useful if we didn't want to surface some properties in the model, yet wanted to set their default values. To set the default value through the store layer, right-click the .edmx file and select Open With ➤ XML Editor. Add `DefaultValue="N/A"` to the `<Property>` tag for the Comment property in the SSDL section of the .edmx file.

Listing 12-7. Handling the SavingChanges event to set the default values

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        using (var context = new EFRecipesEntities())
        {
            context.PurchaseOrders.AddObject(
                            new PurchaseOrder { Amount = 109.98M});
            context.PurchaseOrders.AddObject(
                            new PurchaseOrder { Amount = 20.99M });
            context.PurchaseOrders.AddObject(
                            new PurchaseOrder { Amount = 208.89M});
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            Console.WriteLine("Purchase Orders");
            foreach (var po in context.PurchaseOrders)
            {
                Console.WriteLine("Purchase Order: {0}",
                                po.PurchaseOrderId.ToString(""));
                Console.WriteLine("\tPaid: {0}", po.Paid ? "Yes" : "No");
                Console.WriteLine("\tAmount: {0}", po.Amount.ToString("C"));
                Console.WriteLine("\tCreated On: {0}",
                                po.CreateDate.ToShortTimeString());
                Console.WriteLine("\tModified at: {0}",
                                po.ModifiedDate.ToShortTimeString());
            }
        }
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges += new EventHandler(EFRecipesEntities_SavingChanges);
    }

    void EFRecipesEntities_SavingChanges(object sender, EventArgs e)
    {
        var pos = this.ObjectStateManager
                    .GetObjectStateEntries(EntityState.Added |
                                           EntityState.Modified)
                    .Select(entry => entry.Entity)
                    .OfType<PurchaseOrder>().ToList();
        foreach (var order in pos)
        {
            if (order.EntityState == EntityState.Added)
            {
                order.PurchaseOrderId = Guid.NewGuid();
                order.CreateDate = DateTime.Now;
                order.ModifiedDate = DateTime.Now;
            }
            else if (order.EntityState == EntityState.Modified)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            {
                order.ModifiedDate = DateTime.Now;
            }
        }
    }
}
```

The following is the output from the code in Listing 12-7:

```
Purchase Orders

Purchase Order: 6d07a26e-10f0-4aaa-a65a-2f3eaaef8bf9

        Paid: No

        Amount: $20.99

        Created On: 11:57 AM

        Modified at: 11:57 AM

Purchase Order: 15572f1f-674d-4e3d-a854-551cea412d33

        Paid: No

        Amount: $109.98

        Created On: 11:57 AM

        Modified at: 11:57 AM

Purchase Order: d6c88657-6e72-42e5-9714-cf420f36a403

        Paid: No

        Amount: $208.89

        Created On: 11:57 AM

        Modified at: 11:57 AM
```

How It Works

We demonstrated three different ways to set default values. For values that are static and for which a property is exposed on the entity for the underlying column, we can use the designer's Default Value for

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

the property. This is ideally suited for the Paid property. By default, we want to set this to **false**. New purchase orders are typically unpaid.

For columns that need dynamically calculated values, such as the CreateDate, ModifiedDate, and PurchaseOrderId columns, we wire in a **SavingChanges** event handler that computes these values and sets the column values just before the entity is saved to the database.

Finally, for columns that are not surfaced as properties on the entity and need a static default value, we can use the Default Value attribute in the store layer property definition. In this recipe, we set the comments column default value to "N/A" in the store layer property definition.

There is another option for assigning default values. You could assign them in the constructor for the entity. The constructor is called each time a new instance of the entity is created. This includes each time the instance is materialized from the database. You have to be careful not to overwrite previous values for the properties from the database.

12-8. Retrieving the Original Value of a Property

Problem

You want to retrieve the original value of a property before the entity is saved to the database.

Solution

Let's say you have a model (see Figure 12-11) representing an Employee and part of this entity includes the employee's salary. You have a business rule that an employee's salary cannot be increased by more than 10%. To enforce this rule, you want to check the new salary against the original salary for increases in excess of 10%. You want to do this check just before the entity is saved to the database.

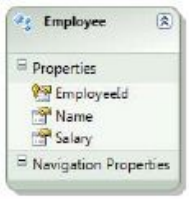


Figure 12-11. *An Employee entity with the employee's salary*

To verify that a salary increase does not exceed 10%, as required by our business rule, we wire in a handler for the **SavingChanges** event. In the handler, we retrieve the current and original values. If the new value is more than 110% of the original value, we throw an exception. This exception, of course, causes the saving of the entity to fail. The code in Listing 12-8 provides the details.

Listing 12-8. Handling the SavingChanges event to enforce the business rule

```
class Program
{
    static void Main(string[] args)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        {
            RunExample();
        }

        static void RunExample()
        {
            using (var context = new EFRecipesEntities())
            {
                var emp1 = new Employee { Name = "Roger Smith", Salary = 108000M };
                var emp2 = new Employee { Name = "Jane Hall", Salary = 81500M };
                context.Employees.AddObject(emp1);
                context.Employees.AddObject(emp2);
                context.SaveChanges();
                emp1.Salary = emp1.Salary * 1.5M;
                try
                {
                    context.SaveChanges();
                }
                catch (Exception)
                {
                    Console.WriteLine("Oops, tried to increase a salary too much!");
                }
            }

            using (var context = new EFRecipesEntities())
            {
                Console.WriteLine();
                Console.WriteLine("Employees");
                foreach (var emp in context.Employees)
                {
                    Console.WriteLine("{0} makes {1}/year", emp.Name,
                                      emp.Salary.ToString("C"));
                }
            }
        }
    }

    public partial class EFRecipesEntities
    {
        partial void OnContextCreated()
        {
            this.SavingChanges += new EventHandler(EFRecipesEntities_SavingChanges);
        }

        void EFRecipesEntities_SavingChanges(object sender, EventArgs e)
        {
            var entries = this.ObjectStateManager
                              .GetObjectStateEntries(EntityState.Modified)
                              .Where(entry => entry.Entity is Employee);
            foreach (var entry in entries)
            {
                var salaryProp = entry.GetModifiedProperties()
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                .FirstOrDefault(p => p == "Salary");
            if (salaryProp != null)
            {
                var originalSalary = Convert.ToDecimal(
                                entry.OriginalValues[salaryProp]);
                var currentSalary = Convert.ToDecimal(
                                entry.CurrentValues[salaryProp]);
                if (originalSalary != currentSalary)
                {
                    if (currentSalary > originalSalary * 1.1M)
                        throw new ApplicationException(
                                 "Can't increase salary more than 10%");
                }
            }
        }
    }
}
```

Licensed by Matt Allen 2175280

The following is the output of the code in Listing 12-8:

```
Oops, tried to increase a salary too much!


Employees

Roger Smith makes $108,000.00/year

Jane Hall makes $81,500.00/year
```

How It Works

In the **SavingChanges** event handler, we first retrieve all the object state entries for the Employee entity that are in the modified state. For each of them, we look for a modified "Salary" property. If we find that the Salary property has been modified, we retrieve both its current value, which represents the value after modification, and its original value. If they differ, we check to see if they differ by more than 10%. If they do, then we throw an **ApplicationException**. Otherwise, we simply return and let Entity Framework save the changes to the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

12-9. Retrieving the Original Association for Independent Associations

Problem

You have an independent association. You want to retrieve the original association prior to saving the changes to the database.

Solution

Suppose you have a model representing an order and the order's status (see Figure 12-12). The fulfillment of an order goes through three stages, as represented in the OrderStatus entity. First, the order is assembled. Next, the order is tested. Finally, the order is shipped. Your application has a business rule that confines all orders to this three-step process. You want to enforce this rule by throwing an exception if an order goes, for example, from assemble to ship without first being tested. The association between Order and OrderStatus is an independent association.

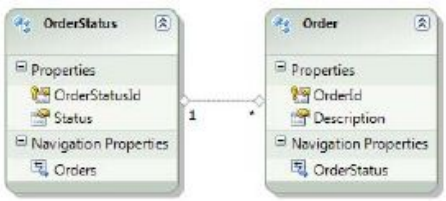


Figure 12-12. *A model with orders and their status*

To solve this problem, we wire in a handler for the **SavingChanges** event. In this handler, we check to verify that the order status changes follow the prescribed sequence. The code in Listing 12-9 provides the details.

Listing 12-9. Enforcing the sequence of fulfillment steps for an order

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            // static order status
            var assemble = new OrderStatus { OrderStatusId = 1,
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                                        Status = "Assemble" };
            var test = new OrderStatus { OrderStatusId = 2,
                                         Status = "Test" };
            var ship = new OrderStatus { OrderStatusId = 3,
                                         Status = "Ship" };
            context.OrderStatus.AddObject(assemble);
            context.OrderStatus.AddObject(test);
            context.OrderStatus.AddObject(ship);

            var order = new Order { Description = "HAL 9000 Supercomputer",
                                    OrderStatus = assemble };
            context.Orders.AddObject(order);
            context.SaveChanges();

            order.OrderStatus = ship;
            try
            {
                context.SaveChanges();
            }
            catch (Exception)
            {
                Console.WriteLine("Oops...better test first.");
            }
            order.OrderStatus = test;
            context.SaveChanges();
            order.OrderStatus = ship;
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            foreach (var order in context.Orders)
            {
                Console.WriteLine("Order {0} [{1}], status = {2}",
                                  order.OrderId.ToString(),
                                  order.Description,
                                  order.OrderStatus.Status);
            }
        }
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges += new EventHandler(EFRecipesEntities_SavingChanges);
    }

    void EFRecipesEntities_SavingChanges(object sender, EventArgs e)
    {
        // all the tracked orders
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        var orders = this.ObjectStateManager.GetObjectStateEntries(
                        EntityState.Modified | EntityState.Unchanged)
                        .Where(entry => entry.Entity is Order)
                        .Select(entry => entry.Entity as Order);

        foreach (var order in orders)
        {
            var deletedEntry = this.ObjectStateManager
                     .GetObjectStateEntries(EntityState.Deleted)
                     .Where(entry => entry.IsRelationship &&
                             entry.EntitySet.Name == order
                                .OrderStatusReference
                                .RelationshipSet.Name).First();
            if (deletedEntry != null)
            {
                EntityKey deletedKey = null;
                if ((EntityKey)deletedEntry.OriginalValues[0] == order.EntityKey)
                {
                    deletedKey = deletedEntry.OriginalValues[1] as EntityKey;
                }
                else if ((EntityKey)deletedEntry.OriginalValues[1] ==
                          order.EntityKey)
                {
                    deletedKey = deletedEntry.OriginalValues[0] as EntityKey;
                }
                if (deletedKey != null)
                {
                    var oldStatus = this.GetObjectByKey(deletedKey)
                                        as OrderStatus;

                    // better be going to the next status
                    if (oldStatus.OrderStatusId + 1 !=
                        order.OrderStatus.OrderStatusId)
                        throw new ApplicationException(
                          "Can't transition to that order status!");
                }
            }
        }
    }
}
```

The following is the output of the code in Listing 12-9:

```
Oops...better test first.

Order 2 [HAL 9000 Supercomputer], status = Ship
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

We wired in a handler for the **SavingChanges** event. In this handler, we picked out the previous order status and the new (current) order status and verified that the new status id is one greater than the previous id. Of course, the code in Listing 12-9 doesn't look quite that simple. Here's how to find both the original order status and the new one.

For independent associations, in the object state manager there is an entry for the order, the order status, and a relationship entry with one end pointing to the order and the other end pointing to the order status. The relationship entry is identified by IsRelationship set to **true**.

First, we get all the orders tracked in the object context. To do this, we use the object state manager to get all the entries that are either modified or unchanged. We use a **Where** clause to filter this down to just entities of type Order.

For each order, we get all object state entries that are deleted. Then we use a **Where** clause to pick out just the relationship entries (IsRelationship is **true**) in the OrderStatus relationship set. Because there should be at most one of these for any order, we pick the first. We look for the deleted relationships because when a relationship is changed, the original one is marked deleted, and the new one is created. Because we're interested in the previous relationship, we look for a deleted relationship between the order and the order status.

Once we have the deleted relationship, we need to look at the original values for the entry to find both the order end and the order status end. Be careful not to reference the current values here. Because the relationship is deleted, referencing the current values will cause an exception. As we don't know which end of the relationship is the order and which end is the order status, we test both.

With the original order status entity in hand, we simply check whether the original OrderStatusId is one less than the new OrderStatusId. We created the OrderStatus objects so that their ids would increment by one just to make the code a little easier.

12-10. Retrieving XML

Problem

You want to treat a scalar property of type string as XML data.

Solution

Let's say you have an XML column in a table in your database. When you import this table into a model, Entity Framework interpreted the data type as a string rather than XML (see Figure 12-13). The current version of Entity Framework does not expose XML data types from the database. You want to work with this property as if it were an XML data type.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 12-13. *A model with a Candidate entity. The Resume property is of type string in the model but of type XML in the database.*

Our Candidate entity's Resume property is of type string in the model, but type XML in the database. To manipulate the property as if it were type XML, we'll make the property private and expose a CandidateResume property as XML.

Select the Resume property and view its properties. Change the setter and getter to private. Next, we need to expose a new property that will surface the resume as XML. The code in Listing 12-10 provides the details.

With the CandidateResume property, we can manipulate the Resume natively by using the XML API. In Listing 12-10, we create a strongly typed resume using XElement class and assign it to the CandidateResume property, which assigns the original string Resume property inside the setter. After saving the Candidate entity to the database, we later update the Resume element inside the CandidateResume and update the changes made to the database.

Listing 12-10. Using the CandidateResume property to expose the resume as XML

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var resume = new XElement("Person",
                new XElement("Name", "Robin St.James"),
                new XElement("Phone", "817 867-5201"),
                new XElement("FirstOffice", "Dog Catcher"),
                new XElement("SecondOffice", "Mayor"),
                new XElement("ThirdOffice", "State Senator"));
            var can = new Candidate { Name = "Robin St.James",
                                      CandidateResume = resume };
            context.Candidates.AddObject(can);
            context.SaveChanges();
            can.CandidateResume.SetElementValue("Phone", "817 555-5555");
            context.SaveChanges();
        }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        using (var context = new EFRecipesEntities())
        {
            foreach (var can in context.Candidates)
            {
                Console.WriteLine("{0}", can.Name);
                Console.WriteLine("Phone: {0}",
                        can.CandidateResume.Element("Phone").Value);
                Console.WriteLine("First Political Office: {0}",
                        can.CandidateResume.Element("FirstOffice").Value);
                Console.WriteLine("Second Political Office: {0}",
                        can.CandidateResume.Element("SecondOffice").Value);
                Console.WriteLine("Third Political Office: {0}",
                        can.CandidateResume.Element("ThirdOffice").Value);
            }
        }
    }
}

public partial class Candidate
{
    private XElement candidateResume = null;

    public XElement CandidateResume
    {
        get
        {
            if (candidateResume == null)
            {
                candidateResume = XElement.Parse(this.Resume);
                candidateResume.Changed += (s, e) =>
                    {
                        this.Resume = candidateResume.ToString();
                    };
            }
            return candidateResume;
        }
        set
        {
            candidateResume = value;
            candidateResume.Changed += (s, e) =>
                {
                    this.Resume = candidateResume.ToString();
                };
            this.Resume = value.ToString();
        }
    }
}
```

The following is the output of the code in Listing 12-10:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Robin St.James

Phone: 817 555-5555

First Political Office: Dog Catcher

Second Political Office: Mayor

Third Political Office: State Senator
```

How It Works

The current release of Entity Framework does not support the XML data type. Given the importance of XML, it is likely that some future version will provide full support. In this recipe, we created a new property, CandidateResume, which exposes the candidate's resume as XML.

The code in Listing 12-10 demonstrates using the CandidateResume property in place of the Resume property. For both the getter and setter, we wired in a handler for the **Changed** event on the XML. This handler keeps the Resume property in sync with the CandidateResume property. Entity Framework will look at the Resume property when it comes time to persist an instance of the Candidate entity. Only changes to the Resume property will be saved. We need to reflect changes in the CandidateResume property to the Resume property for the database to stay in sync (via Entity Framework).

12-11. Applying Server-Generated Values to Properties

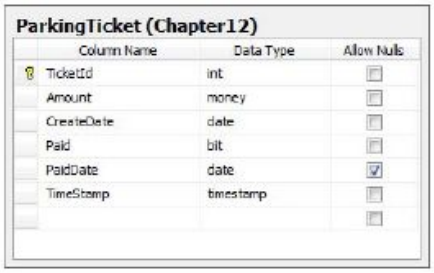
Problem

You have several columns in a table whose values are generated by the database. You want to have Entity Framework set the corresponding entity properties after inserts and updates.

Solution

Suppose you have a table like the one in Figure 12-14.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



ParkingTicket (Chapter12)

Column Name	Data Type	Allow Nulls
TicketId	int	☐
Amount	money	☐
CreateDate	date	☐
Paid	bit	☐
PaidDate	date	☑
TimeStamp	timestamp	☐
		☐

Figure 12-14. *The ParkingTicket table with the TicketId, CreateDate, PaidDate, and TimeStamp columns generated by the database*

Also, let's say you have created a trigger, like the one in Listing 12-11, so that the PaidDate column is populated when the Paid column is set to true. You've also set the TicketId to be an Identity column and CreateDate to default to the current date. With the trigger in Listing 12-11 and the automatically generated values, only the Amount and Paid columns are required for an insert.

Listing 12-11. A trigger that sets the PaidDate column when the Paid bit is set to ***true***

```
create trigger Chapter12.UpdateParkingTicket on Chapter12.ParkingTicket
for update
as
update Chapter12.ParkingTicket
  set PaidDate = getdate()
  from Chapter12.ParkingTicket
    join inserted i on ParkingTicket.TicketId = i.TicketId
  where i.Paid = 1
```

After an insert or an update, you want Entity Framework to populate the entity with the values generated by the database. To create the model that supports this, do the following:

1. Right-click the project and select Add ➤ New Item. Add a new ADO.NET Entity Data Model. Import the ParkingTicket table. The resulting model should look like the one shown in Figure 12-15.
2. Right-click on each of the scalar properties in the ParkingTicket entity. View the properties of each. Notice that the StoreGeneratedPattern property is set to Identity for the TicketId. For CreateDate and TimeStamp the StoreGeneratedPattern property is set to Computed. The StoreGeneratedPattern property for PaidDate is not set. Change this value to Computed.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

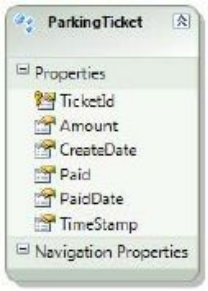


Figure 12-15. The model with the ParkingTicket entity

Listing 12-12. Code to check if the database generated values are populated back to the properties on inserts and updates

```
using (var context = new EFRecipesEntities())
{
    context.ParkingTickets.AddObject(new ParkingTicket
                        { Amount = 132.0M, Paid = false });
    context.ParkingTickets.AddObject(new ParkingTicket
                        { Amount = 255.0M, Paid = false });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    foreach (var ticket in context.ParkingTickets)
    {
        Console.WriteLine("Ticket: {0}", ticket.TicketId);
        Console.WriteLine("Date: {0}", ticket.CreateDate.ToShortDateString());
        Console.WriteLine("Amount: {0}", ticket.Amount.ToString("C"));
        Console.WriteLine("Paid: {0}",
              ticket.PaidDate.HasValue ?
                ticket.PaidDate.Value.ToShortDateString() : "Not Paid");
        Console.WriteLine();
        ticket.Paid = true; // just paid ticket!
    }

    // save all those Paid flags
    context.SaveChanges();
    foreach (var ticket in context.ParkingTickets)
    {
        Console.WriteLine("Ticket: {0}", ticket.TicketId);
        Console.WriteLine("Date: {0}", ticket.CreateDate.ToShortDateString());
        Console.WriteLine("Amount: {0}", ticket.Amount.ToString("C"));
        Console.WriteLine("Paid: {0}",
              ticket.PaidDate.HasValue ?
                ticket.PaidDate.Value.ToShortDateString() : "Not Paid");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        Console.WriteLine();
    }
}
```

The following is the output of the code in Listing 12-12:

```
Ticket: 5

Date: 3/24/2010

Amount: $132.00

Paid: Not Paid


Ticket: 6

Date: 3/24/2010

Amount: $255.00

Paid: Not Paid


Ticket: 5

Date: 3/24/2010

Amount: $132.00

Paid: 3/24/2010


Ticket: 6

Date: 3/24/2010

Amount: $255.00

Paid: 3/24/2010
```

Licensed by
Matt Allen
2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

When you set a property's StoreGeneratedPattern to Identity or Computed, Entity Framework knows that the database will generate the value. Entity Framework will retrieve these columns from the database with a subsequent select statement.

When the StoreGeneratePattern is set to Identity, Entity Framework retrieves the database generated value just once at the time of insert. When the StoreGeneratedPattern is set to Computed, Entity Framework will refresh the value on each insert and update. In this example, the PaidDate column was set by the trigger (because we set Paid to `true`) on update and Entity Framework acquired this value after the update.

12-12. Validating Entities on SavingChanges

Problem

You want to validate entities before they are saved to the database.

Solution

Suppose you have a model like the one shown in Figure 12-16.

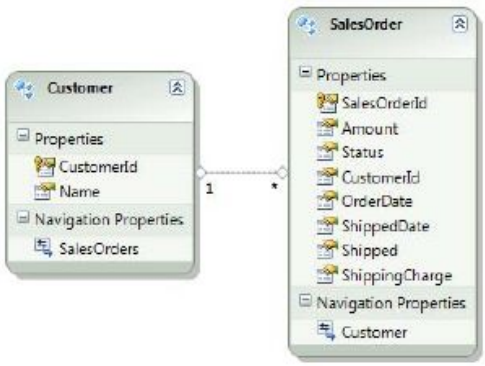


Figure 12-16. A model for customers and their orders

There are certain business rules around customers and their orders. You want to make sure that these rules are checked before an order is saved to the database. Let's say you have the following rules:

- The order date on an order must be after the current date
- The ship date on an order must be after the order date
- An order cannot be shipped unless it is in an "Approved" status

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

- If an order amount is over $5,000 there is not shipping charge
- An order that has shipped cannot be deleted

To check if changes to an entity violates these rules, we'll define an IValidator interface that has just one method: **Validate()**. Any of our entity types can implement this interface. For this example, we'll show the implementation for the SalesOrder entity. We'll handle the SavingChanges event and call Validate() on all entities that implement IValidator. This will allow us to intercept and validate entities before they are saved to the database. The code in Listing 12-13 provides the details.

Listing 12-13. Validating SaleOrder entities in the SavingChanges event

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        // bad order date
        using (var context = new EFRecipesEntities())
        {
            var customer = new Customer { Name = "Phil Marlowe" };
            var order = new SalesOrder { OrderDate = DateTime.Parse("3/12/18"),
                                         Amount = 19.95M, Status = "Approved",
                                         ShippingCharge = 3.95M,
                                         Customer = customer };
            context.Customers.AddObject(customer);
            try
            {
                context.SaveChanges();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }

        // order shipped before it was ordered
        using (var context = new EFRecipesEntities())
        {
            var customer = new Customer { Name = "Phil Marlowe" };
            var order = new SalesOrder { OrderDate = DateTime.Parse("3/12/10"),
                                         Amount = 19.95M, Status = "Approved",
                                         ShippingCharge = 3.95M,
                                         Customer = customer };
            context.Customers.AddObject(customer);
            context.SaveChanges();
            try
            {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            order.Shipped = true;
            order.ShippedDate = DateTime.Parse("3/10/10");
            context.SaveChanges();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    // order shipped, but not approved
    using (var context = new EFRecipesEntities())
    {
        var customer = new Customer { Name = "Phil Marlowe" };
        var order = new SalesOrder { OrderDate = DateTime.Parse("3/12/10"),
                                     Amount = 19.95M, Status = "Pending",
                                     ShippingCharge = 3.95M,
                                     Customer = customer };
        context.Customers.AddObject(customer);
        context.SaveChanges();
        try
        {
            order.Shipped = true;
            order.ShippedDate = DateTime.Parse("3/13/10");
            context.SaveChanges();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
    }

    // order over $5,000 and shipping not free
    using (var context = new EFRecipesEntities())
    {
        var customer = new Customer { Name = "Phil Marlowe" };
        var order = new SalesOrder { OrderDate = DateTime.Parse("3/12/10"),
                                     Amount = 6200M, Status = "Approved",
                                     ShippingCharge = 59.95M,
                                     Customer = customer };
        context.Customers.AddObject(customer);
        context.SaveChanges();
        try
        {
            order.Shipped = true;
            order.ShippedDate = DateTime.Parse("3/13/10");
            context.SaveChanges();
        }
        catch (Exception ex)
        {
            Console.WriteLine(ex.Message);
        }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }

        // order deleted after it was shipped
        using (var context = new EFRecipesEntities())
        {
            var customer = new Customer { Name = "Phil Marlowe" };
            var order = new SalesOrder { OrderDate = DateTime.Parse("3/12/10"),
                                         Amount = 19.95M, Status = "Approved",
                                         ShippingCharge = 3.95M,
                                         Customer = customer };
            context.Customers.AddObject(customer);
            context.SaveChanges();
            order.Shipped = true;
            order.ShippedDate = DateTime.Parse("3/13/10");
            context.SaveChanges();
            try
            {
                context.DeleteObject(order);
                context.SaveChanges();
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges +=new EventHandler(EFRecipesEntities_SavingChanges);
    }

    private void EFRecipesEntities_SavingChanges(object sender, EventArgs e)
    {
        var entries = this.ObjectStateManager
                      .GetObjectStateEntries(EntityState.Added |
                                             EntityState.Modified |
                                             EntityState.Deleted)
                      .Where(entry => entry.Entity is IValidator)
                      .Select(entry => entry).ToList();
        foreach (var entry in entries)
        {
            var entity = entry.Entity as IValidator;
            entity.Validate(entry);
        }
    }
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
public interface IValidator
{
    void Validate(ObjectStateEntry entry);
}

public partial class SalesOrder : IValidator
{
    public void Validate(ObjectStateEntry entry)
    {
        if (entry.State == EntityState.Added)
        {
            if (this.OrderDate > DateTime.Now)
                throw new ApplicationException(
                  "OrderDate cannot be after the current date");
        }
        else if (entry.State == EntityState.Modified)
        {
            if (this.ShippedDate < this.OrderDate)
            {
                throw new ApplicationException(
                  "ShippedDate cannot be before OrderDate");
            }
            if (this.Shipped.Value && this.Status != "Approved")
            {
                throw new ApplicationException(
                  "Order cannot be shipped unless it is Approved");
            }
            if (this.Amount > 5000M && this.ShippingCharge != 0)
            {
                throw new ApplicationException(
                  "Orders over $5000 ship for free");
            }
        }
        else if (entry.State == EntityState.Deleted)
        {
            if (this.Shipped.Value)
                throw new ApplicationException(
                  "Shipped orders cannot be deleted");
        }
    }
}
```

The following is the output of the code in Listing 12-13:

```
OrderDate cannot be after the current date

ShippedDate cannot be before OrderDate

Order cannot be shipped unless it is Approved
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Orders over $5000 ship for free

Shipped orders cannot be deleted
```

How It Works

When you call **SaveChanges()**, Entity Framework raises the SavingChanges event before it saves the object changes to the database. We implemented the partial method **OnContextCreated()** so that we can wire in a handler for this event. When SavingChanges is raised, we handle the event by calling the **Validate()** method on every entity that implements the IValidator interface. We've shown an implementation of this interface that supports our business rules. If you have business rules for other entity types in your model, you could implement the IValidator interface for them.

Best Practice

Business rules in many applications almost always change over time. Industry or government regulation changes, continuous process improvement programs, evolving fraud prevention and many other factors influence the introduction of new business rules and changes to existing rules. It's a best practice to organize your code base so that the concerns around business rule validation and enforcement are more easily maintained. Often this means keeping all of this code in a separate assembly or in a separate folder in the project. Defining and implementing interfaces, such as the IValidator interface in this recipe, help to ensure that business rules validation is uniformly applied.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 13

Improving Performance

The recipes in this chapter cover a wide range of specific ways to improve the performance of your Entity Framework applications. In many cases, simple changes to a query, changes to the model, or even pushing startup overhead to a different part of application can significantly improve some aspect of your application's performance.

13-1. Optimizing Queries in a Table per Type Inheritance Model

Problem

You want to improve the performance of a query in a model with Table per Type inheritance.

Solution

Let's say you have a simple Table per Type inheritance model like the one shown in Figure 13-1.

Figure 13-1. *A simple Table per Type inheritance model for Salaried and Hourly employees*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

You want to query this model for a given employee. To improve the performance of the query if you know the type of employee, use the **OfType<T>()** operator to narrow the result to entities of the specific type, as shown in Listing 13-1.

Listing 13-1. Improving the performance of a query against a Table per Type inheritance model if you know the entity type

```
using (var context = new EFRecipesEntities())
{
    context.Employees.AddObject(new SalariedEmployee { Name = "Robin Rosen",
                                    Salary = 89900M });
    context.Employees.AddObject(new HourlyEmployee { Name = "Steven Fuller",
                                    Rate = 11.50M });
    context.Employees.AddObject(new HourlyEmployee { Name = "Karen Steele",
                                    Rate = 12.95m });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // a typical way to get Steven Fuller's entity
    var emp1 = context.Employees.Single(e => e.Name == "Steven Fuller");
    Console.WriteLine("{0}'s rate is: {1} per hour", emp1.Name,
                      ((HourlyEmployee)emp1).Rate.ToString("C"));

    // slightly more efficient way if we know that Steven is an HourlyEmployee
    var emp2 = context.Employees.OfType<HourlyEmployee>()
                                .Single(e => e.Name == "Steven Fuller");
    Console.WriteLine("{0}'s rate is: {1} per hour", emp2.Name,
                      emp2.Rate.ToString("C"));
}
```

The following is the output of the code in Listing 13-1:

```
Steven Fuller's rate is: $11.50 per hour
Steven Fuller's rate is: $11.50 per hour
```

How It Works

The key to making the query in a Table per Type inheritance model more efficient is to explicitly tell Entity Framework the type of the expected result. This allows Entity Framework to generate code that limits the search to the specific tables that hold the values for the base type and the derived type. Without this information, Entity Framework has to generate a query that pulls together all the results from all the tables holding derived type values and then determines the appropriate type for materialization. Depending on the number of derived types and the complexity of your model, this may require substantially more work than necessary. Of course, this assumes that you know exactly what derived type the query will return.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

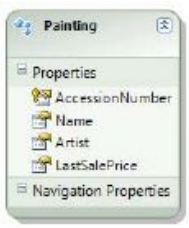
13-2. Retrieving a Single Entity Using an Entity Key

Problem

You want to retrieve a single entity using an entity key.

Solution

Suppose you have a model with an entity type representing a painting. The model might look like the one in Figure 13-2.



Licensed by Matt Allen 21275280

Figure 13-2. *The Painting entity type in our model*

To retrieve a particular entity using an entity key can be more efficient than using a **where** clause. To retrieve an entity using an entity key, use the **GetObjectByKey()** method, as shown in Listing 13-2.

Listing 13-2. Retrieving an entity using an entity key

```
using (var context = new EFRecipesEntities())
{
    context.Paintings.AddObject(new Painting { AccessionNumber = "PN001",
                                Name = "Sunflowers",
                                Artist = "Rosemary Golden",
                                LastSalePrice = 1250M });
    context.Paintings.AddObject(new Painting { AccessionNumber = "PN002",
                                Name = "Red River",
                                Artist = "Alex Jones",
                                LastSalePrice = 1800M });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // let's assume we already know the key for the painting
    var p = context.GetObjectByKey(
                new System.Data.EntityKey(
                          "EFRecipesEntities.Paintings",
                          "AccessionNumber", "PN001"));
    Painting painting = (Painting) p;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    Console.WriteLine("The painting with accession number {0}",
                      painting.AccessionNumber);
    Console.WriteLine("\tName: {0}", painting.Name);
    Console.WriteLine("\tArtist: {0}", painting.Artist);
    Console.WriteLine("\tSale Price: {0}",
                      painting.LastSalePrice.ToString("C"));
}
```

The following is the output of the code in Listing 13-2:

```
The painting with accession number PN001

        Name: Sunflowers

        Artist: Rosemary Golden

        Sale Price: $1,250.00
```

How It Works

At first, using the **GetObjectByKey()** method seems like a lot more work than using a simple LINQ expression with a **where** clause. The call to **GetObjectByKey()** is more noisy, and the returned object must be cast to our entity type. The one advantage that **GetObjectByKey()** has is that it can retrieve the entity from the object context without making a round trip to the database. Using a LINQ expression with a **where** clause and a **first()** or **single()** method call will always cause a round trip to the database even if the entity has been previously retrieved and is in the object context.

Bypassing the database and retrieving the entity directly from the object context is the primary performance benefit we might see from the **GetObjectByKey()** method. This assumes, of course, that some previous operation has already fetched the entity from the database and placed it into the object context.

GetObjectByKey() will throw an exception if an entity with the given entity key cannot be found in either the object context or in the database. The **TryGetObjectByKey()** method does the same thing as the **GetObjectByKey()** method but will not throw an exception if the entity can't be found.

There are a couple of important things to remember about the **GetObjectByKey()** method. First, it won't return entities from the object context if they are in the inserted state. However, it will return objects that are marked for deletion.

If **MergeOption.NoTracking** is set, calls to **GetObjectByKey()** will make a round trip to the database because the **MergeOption.NoTracking** will keep objects from being inserted into the object context after they are materialized. If the objects are not in the object context, then **GetObjectByKey()** has no choice but to retrieve the objects from the database.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

13-3. Retrieving Entities for Read Only

Problem

You want to efficiently retrieve some entities that you will display only and not need to update.

Solution

A very common activity in many applications, especially websites, is to let the user browse through data. In many cases, the user will never update the data. For these situations, we can make our code a lot more efficient if we avoid the management overhead of the object context. We can do this using the **NoTracking** merge option.

Let's say you have an application that manages appointments for doctors. Your model may look something like the one in Figure 13-3.

Figure 13-3. *A model for managing doctors and their appointments*

To retrieve the doctors and the companies they work for without adding them to the object context, set the MergeOption to **MergeOption.NoTracking** as we have in Listing 13-3.

Listing 13-3. Doing a simple query using the ***NoTracking*** *merge option*

```
using (var context = new EFRecipesEntities())
{
    var company = new Company { Name = "Paola Heart Center" };
    var doc1 = new Doctor { Name = "Jill Mathers", Company = company };
    var doc2 = new Doctor {Name = "Robert Stevens", Company = company};
    var app1 = new Appointment { AppointmentDate = DateTime.Parse("3/18/2010"),
                                 Patient = "Karen Rodgers", Doctor = doc1 };
    var app2 = new Appointment { AppointmentDate = DateTime.Parse("3/20/2010"),
                                 Patient = "Steven Cook", Doctor = doc2 };
    context.Companies.AddObject(company);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Appointments for Doctors...");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.Doctors.MergeOption = MergeOption.NoTracking;
    var doctors = context.Doctors.Include("Company");
    foreach (var doctor in doctors)
    {
        Console.WriteLine("Doctor: {0} [{1}]", doctor.Name, doctor.Company.Name);
        Console.WriteLine("Appointments: {0}",
                          doctor.Appointments.Count().ToString());
    }
}
```

The following is the output of the code in Listing 13-3:

```
Appointments for Doctors...

Doctor: Jill Mathers [Paola Heart Center]

Appointments: 1

Doctor: Robert Stevens [Paola Heart Center]

Appointments: 1
```

How It Works

ObjectSet<T> and **ObjectQuery<T>** expose a **MergeOption** property. This property can be set to a number of different values including the **NoTracking** option. With **NoTracking**, the objects resulting from a query are not tracked in the object context. In our case, this includes the doctors and the companies the doctors work for because our query explicitly included these. If we were to later load related entities such as the appointments, they also would be excluded from the object context.

The default merge option is **AppendOnly**. This is why, by default, the results of our queries are tracked in the object context. This makes updating and deleting objects effortless, but at the cost of some memory overhead. For applications that stream large numbers of objects, such as browsing products at an ecommerce web site, using the **NoTracking** option often results in less resource overhead and better application performance.

There are a couple of things to note about the **NoTracking** option. If you retrieve an entity with **NoTracking** on and later need to update the entity, you can simply **Attach()** it to the object context before you make any changes.

If your model is using foreign key associations, the foreign keys are tracked as part of the associations and are always returned regardless of the current merge option value. However, the EntityReference.EntityKey values will be null when you use **NoTracking**. In our example, the CompanyId properties will have meaning values, but the CompanyReference.EntityKey values will be null.

When you set a tracking option on an **ObjectSet<T>** (as we do in Listing 13-3), the merge option affects all future queries against the object set. If you set a merge option on an **ObjectQuery<T>**, it will not affect the merge option of the underlying object set.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

13-4. Improving the Startup Time

Problem

You want to improve the startup time of the model.

Solution

When you execute a query against a model, Entity Framework converts the model into a collection of Entity SQL views. These views contain all the necessary Entity SQL to query the store model directly. Once these views are created, they are cached and reused during subsequent requests. These views can be created at design time, which eliminates the need to create them at runtime during a cold startup.

Let's say you have a model that looks like the one in Figure 13-4.

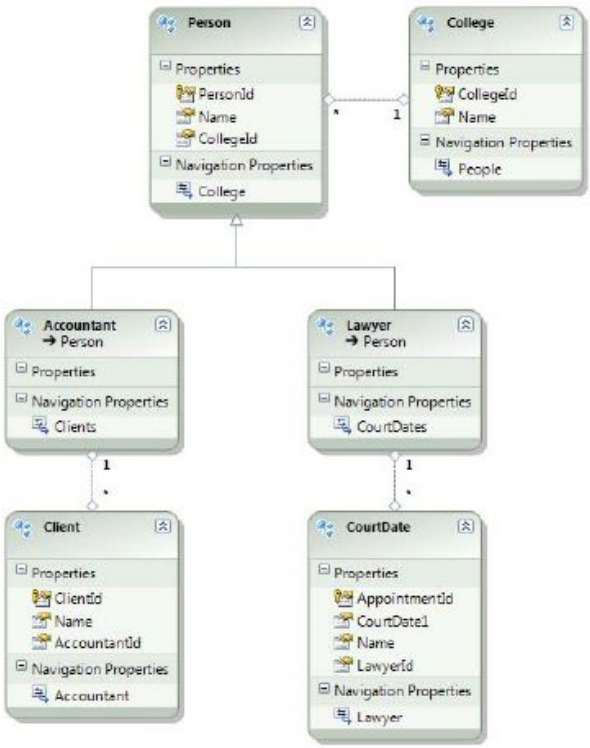


Figure 13-4. *A relatively simple model that can benefit from pregenerated views*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

To pregenerate the views for this model, we'll use a T4 template. We can write the T4 template, but the Entity Framework development team has already created a sample T4 template that we can use.

To start off, we need to get a copy of the T4 template created by the Entity Framework development team. We include the file CSharp.Views.tt in our example download available from Apress.com. (You can download the example code in a zip file, from the book's catalog page at **http://apress.com/book/view/1430227036**. Look for the "Source Code" link under the book's cover image.) Use CSharp.Views.tt as your T4 template in this recipe. To use this template and pregenerate the views, do the following:

1. Copy the CSharp.Views.tt template file into the same directory as the model. This is the directory that contains the .edmx file.
2. Rename the template to match the name of the .edmx file. In our case, rename the file to **Recipe4.Views.tt.**
3. Add the Recipe4.Views.tt template file to the project. Once it is part of the project, right-click the Recipe4.Views.tt file in the Solution Explorer and select Run Custom Tool. This will create a C# file beneath Recipe4.Views.tt. If you generate the file and view its properties, you will notice that the Build Action is set to Compile. This generated code will be built as part of the project.
4. Right-click the Recipe4.Views.tt file in the Solution Explorer and select Run Custom Tool. This will generate the corresponding Recipe4.Views.cs files to be generated from the template.

The build action for the Recipe4.Views.cs files is set to Compile. When your project builds, this file, which contains the generated views, will be built as part of your project.

How It Works

The first use of an object context in an application domain (**AppDomain**) causes the views for the model to be generated if they don't already exist in the application domain. View generation, although not often expensive, does impose some initial startup cost. For larger models, this startup cost can be significant. In this recipe, we generated the code for the views using a T4 template. Because this precompiled view is part of the project and present in the application domain, no view creation is required. This eliminates the initial startup costs.

To demonstrate the effects of precompiling the views for the simple model in Figure 13-4, we used the code in Listing 13-4 to time the cost of an initial query. Here we ran the code ten times with and without the precompiled views. Because the views are cached for the application domain, we actually ran the test application ten times for each scenario rather than putting it in a loop for a single run. Our results are shown in Table 13-1.

Listing 13-4. Measuring the execution time of a simple query against our model

```
static void GetTimes()
{
    using (var context = new EFRecipesEntities())
    {
        var stopwatch = new Stopwatch();
        stopwatch.Start();
        var lawyer = context.People.Include("College")
                        .OfType<Lawyer>().Include("CourtDates").First();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        stopwatch.Stop();
        Console.WriteLine("Execution Time: {0}", stopwatch.ElapsedMilliseconds);
    }
}
```

Table 13-1. *Runtimes with and without Precompiled Views*

Run	Without Precompiled Views (ms)	With Precompiled Views (ms)
1	419	388
2	420	394
3	416	387
4	412	388
5	413	388
6	413	386
7	419	385
8	418	387
9	414	393
10	414	387
Average	415	388.3

For the ten runs, the average time to execute the query, including the view compile time, was 415 ms. The average for the same query after ten runs with the views precompiled from the code generated by the T4 template was 388.3 ms. For this relatively simple model, precompiling the views saved, on average, a little more than 26 ms. For more complex models, we would expect significantly longer startup times without precompiling and more savings with precompiling.

The other way to generate views for a model is to use the EdmGen.exe utility. This can be a little tedious because EdmGen.exe requires three separate files representing the storage, conceptual, and mapping layers from the .edmx file.

13-5. Efficiently Building a Search Query

Problem

You want to write a search query using LINQ so that it is translated to more efficient SQL.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Let's say you have a model like the one in Figure 13-5.

Figure 13-5. *A simple model with a Reservation entity*

You want to write a search query using LINQ to find reservations for a particular person or reservations on a given date or both. You might use the **let** keyword as we did in the first query in Listing 13-5 to make the LINQ expression fairly clean and easy to read. However, the **let** keyword is translated to more complex and often less efficient SQL. Instead of using the **let** keyword, consider explicitly creating two conditions in the **where** clause, as we did in the second query in Listing 13-5.

Listing 13-5. Using both the ***let*** *keyword and explicit conditions in the query*

```
using (var context = new EFRecipesEntities())
{
    context.Reservations.AddObject(new Reservation { Name = "James Jordan",
         ResDate = DateTime.Parse("4/18/10")});
    context.Reservations.AddObject(new Reservation { Name = "Katie Marlowe",
         ResDate = DateTime.Parse("3/22/10")});
    context.Reservations.AddObject(new Reservation { Name = "Roger Smith",
         ResDate = DateTime.Parse("4/18/10")});
    context.Reservations.AddObject(new Reservation { Name = "James Jordan",
         ResDate = DateTime.Parse("5/12/10") });
    context.Reservations.AddObject(new Reservation { Name = "James Jordan",
         ResDate = DateTime.Parse("6/22/10") });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    DateTime? searchDate = null;
    string searchName = "James Jordan";

    Console.WriteLine("More complex SQL...");
    var query2 = from reservation in context.Reservations
                 let dateMatches = searchDate == null ||
                     reservation.ResDate == searchDate
                 let nameMatches = searchName == string.Empty ||
                     reservation.Name.Contains(searchName)
                 where dateMatches && nameMatches
                 select reservation;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var reservation in query2)
    {
        Console.WriteLine("Found reservation for {0} on {1}", reservation.Name,
                          reservation.ResDate.ToShortDateString());
    }

    Console.WriteLine("Cleaner SQL...");
    var query1 = from reservation in context.Reservations
                 where (searchDate == null ||
                        reservation.ResDate == searchDate)  &&
                       (searchName == string.Empty ||
                        reservation.Name.Contains(searchName))
                 select reservation;
    foreach (var reservation in query1)
    {
        Console.WriteLine("Found reservation for {0} on {1}", reservation.Name,
                          reservation.ResDate.ToShortDateString());
    }
}
```

The following is the output of the code in Listing 13-5:

```
More complex SQL...

Found reservation for James Jordan on 4/18/2010

Found reservation for James Jordan on 5/12/2010

Found reservation for James Jordan on 6/22/2010

Cleaner SQL...

Found reservation for James Jordan on 4/18/2010

Found reservation for James Jordan on 5/12/2010

Found reservation for James Jordan on 6/22/2010
```

How It Works

Writing conditions inline, as we did in the second query in Listing 13-5, is not very good for readability or maintainability. Typically, we would use the **let** keyword to make the code cleaner and more readable. In some cases, however, this leads to more complex and often less efficient SQL code.

Let's take a look at the SQL generated by both approaches. Listing 13-6 shows the SQL generated for the first query. Notice that the **where** clause contains a **case** statement with quite a bit of **cast**'ing going on. If we had more parameters in our search query, beyond just name and reservation date, the resulting SQL statement would get even more complicated.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 13-7 shows the SQL generated from the second query, where we created the conditions inline. This query is simpler and likely more efficient at runtime.

Listing 13-6. SQL generated when let is used in the LINQ query

```
SELECT
[Extent1].[ReservationId] AS [ReservationId],
[Extent1].[ResDate] AS [ResDate],
[Extent1].[Name] AS [Name]
FROM [Chapter13].[Reservation] AS [Extent1]
WHERE (
  (CASE WHEN (@p__linq__0 IS NULL OR
         @p__linq__1 =  CAST( [Extent1].[ResDate] AS datetime2))
        THEN cast(1 as bit)
        WHEN ( NOT (@p__linq__0 IS NULL OR
         @p__linq__1 =  CAST( [Extent1].[ResDate] AS datetime2)))
        THEN cast(0 as bit) END) = 1) AND
  ((CASE WHEN ((@p__linq__2 = @p__linq__3) OR
          ([Extent1].[Name] LIKE @p__linq__4 ESCAPE N''~''))
         THEN cast(1 as bit)
         WHEN ( NOT ((@p__linq__2 = @p__linq__3) OR
          ([Extent1].[Name] LIKE @p__linq__4 ESCAPE N''~'')))
         THEN cast(0 as bit) END) = 1)
```

Listing 13-7. Cleaner, more efficient SQL generated when not using ***let*** *in a LINQ query*

```
SELECT
[Extent1].[ReservationId] AS [ReservationId],
[Extent1].[ResDate] AS [ResDate],
[Extent1].[Name] AS [Name]
FROM [Chapter13].[Reservation] AS [Extent1]
WHERE (@p__linq__0 IS NULL OR
       @p__linq__1 = CAST( [Extent1].[ResDate] AS datetime2)) AND
      ((@p__linq__2 = @p__linq__3) OR
       ([Extent1].[Name] LIKE @p__linq__4 ESCAPE N''~''))
```

13-6. Making Change Tracking with POCO Faster

Problem

You are using POCO and want to improve the performance of change tracking.

Solution

Suppose you have the model with an account and related payments like the one in Figure 13-6.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 13-6. *A model with an Account entity and a related Payment*

If you are using POCO and want to get the best change-tracking performance, we need to have Entity Framework create change-tracking proxies for our classes so that it is immediately notified of changes to our objects. To get the change-tracking proxies created for our classes, mark each property as **`virtual`**, as shown in Listing 13-8. Additionally, each navigation property is of type **`ICollection<T>`**.

Listing 13-8. Our POCO classes with properties marked as ***`virtual`*** *and the navigation properties are of type ICollection<T>*

Licensed by Matt Allen 21735280

```
public class Account
{
    public virtual int AccountId { get; set; }
    public virtual string Name { get; set; }
    public virtual decimal Balance { get; set; }
    public virtual ICollection<Payment> Payments { get; set; }
}

public class Payment
{
    public virtual int PaymentId { get; set; }
    public virtual string Name { get; set; }
    public virtual string PaidTo { get; set; }
    public virtual decimal Paid { get; set; }
    public virtual int AccountId { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    private ObjectSet<Account> _accounts;
    private ObjectSet<Payment> _payments;

    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
        _accounts = CreateObjectSet<Account>();
        _payments = CreateObjectSet<Payment>();
    }

    public ObjectSet<Account> Accounts
    {
        get { return _accounts; }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    }

    public ObjectSet<Payment> Payments
    {
        get { return _payments; }
    }
}
```

The code in Listing 13-9 illustrates inserting, retrieving, and updating our model. Note that we use the **CreateObject()** method on the object context to get the proxies for our POCO classes.

Listing 13-9. Inserting, retrieving, and updating our model

```
using (var context = new EFRecipesEntities())
{
    Stopwatch watch = new Stopwatch();
    watch.Start();
    for (int i = 0; i < 5000; i++)
    {
        var account = context.CreateObject<Account>();
        account.Name = "Test" + i.ToString();
        account.Balance = 10M;
        account.Payments.Add(new Payment {
             PaidTo = "Test" + (i + 1).ToString(), Paid = 5M });
        context.Accounts.AddObject(account);
    }
    context.SaveChanges();
    watch.Stop();
    Console.WriteLine("Time to insert: {0} seconds",
             watch.Elapsed.TotalSeconds.ToString());
}

using (var context = new EFRecipesEntities())
{
    Stopwatch watch = new Stopwatch();
    watch.Start();
    var accounts = context.Accounts.Include("Payments").ToList();
    watch.Stop();
    Console.WriteLine("Time to read: {0} seconds",
             watch.Elapsed.TotalSeconds.ToString());

    watch.Restart();
    foreach (var account in accounts)
    {
        account.Balance += 10M;
        account.Payments.First().Paid += 1M;
    }
    context.SaveChanges();
    watch.Stop();
    Console.WriteLine("Time to update: {0} seconds",
            watch.Elapsed.TotalSeconds.ToString());
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Change tracking with POCO occurs using either snapshots or proxies. With snapshots, Entity Framework maintains the state, or snapshot, of the values and relationships before changes are made. This snapshot is used to compare values after changes have been made to determine which properties on which objects have changed. For this approach, Entity Framework maintains two copies of each object so that it can determine what needs to happen when **SaveChanges()** or **DetectChanges()** is called on the object context. You might expect this approach to be very slow, but Entity Framework is very fast in finding the differences.

In the second approach, a proxy that implements the IEntityWithChangeTracking interface is created for each POCO object. This proxy is responsible for notifying the object state manager of changes to values and relationships on the object. Entity Framework creates these proxies for your POCO object when you mark all the properties on your POCO class as **virtual**. Navigation properties that return a collection must return **ICollection<T>**. Proxies avoid the potentially complex object by object comparisons of the snapshot approach. It does, however, require some overhead to track each change as it occurs.

Although change tracking with proxies immediately notifies the object state manager about changes to the objects and avoids object comparisons, in practice, the performance benefits are typically seen only when the model is quite complex and/or when few changes are made to a large number of objects. The model in Figure 13-6 is very simple, and every object is updated in code in Listing 13-9. If you change the code to use snapshots, you will notice only a second or so saved for the updates when proxies are used.

13-7. Compiling LINQ Queries

Problem

You want to improve the performance of queries that are reused several times.

Solution

Let's say you have a model like the one in Figure 13-7.

Figure 13-7. *A model with an Associate and its related Paycheck*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In this model, each Associate has zero or more Paychecks. You have a LINQ query that is used repeatedly throughout your application and you want to improve the performance of this query by compiling it just once and reusing the compiled version in subsequent executions.

To compile a query, use the **CompileQuery.Compile()** method. There are more than a dozen overloads for this method. In Listing 13-10, we illustrate the most basic use in which the generic takes an object context, zero or more parameters, and a return type.

To illustrate the performance benefit, we've instrumented the code in Listing 13-10 to print the number of ticks for each of ten iterates taken for both the uncompiled and compiled versions of the LINQ query. In this query, we can see that we get roughly a 3x performance boost. Most of this, of course, is due to the relatively high cost of compiling versus the low cost for actually performing this simple query.

Listing 13-10. Comparing the performance of a simple compiled LINQ query

```
using (var context = new EFRecipesEntities())
{
    var a1 = new Associate { Name = "Robert Stevens", City = "Raytown" };
    a1.Paychecks.Add(new Paycheck { PayDate = DateTime.Parse("3/1/10"),
                                    Gross = 1802.83M });
    a1.Paychecks.Add(new Paycheck { PayDate = DateTime.Parse("3/15/10"),
                                    Gross = 1924.91M });
    var a2 = new Associate { Name = "Karen Thorp", City = "Gladstone" };
    a2.Paychecks.Add(new Paycheck { PayDate = DateTime.Parse("3/1/10"),
                                    Gross = 2102.34M });
    a2.Paychecks.Add(new Paycheck { PayDate = DateTime.Parse("3/15/10"),
                                    Gross = 1992.18M });
    context.Associates.AddObject(a1);
    context.Associates.AddObject(a2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Stopwatch watch = new Stopwatch();
    long totalTicks = 0;

    // warm things up
    context.Associates.Where(a => a.Name.StartsWith("Karen")).ToList();

    // query gets compiled each time
    for (int i = 0; i < 10; i++)
    {
        watch.Restart();
        context.Associates.Where(a => a.Name.StartsWith("Karen")).ToList();
        watch.Stop();
        totalTicks += watch.ElapsedTicks;
        Console.WriteLine("Not Compiled: {0}", watch.ElapsedTicks.ToString());
    }
    Console.WriteLine("Average ticks without compiling: {0}",
                      (totalTicks / 10).ToString());
    Console.WriteLine("");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    // compile the query just once and re-use
    var query = CompiledQuery.Compile<EFRecipesEntities, IQueryable<Associate>>
                  (ctx =>
                     from a in ctx.Associates
                     where a.Name.StartsWith("Karen")
                     select a);
    totalTicks = 0;
    for (int i = 0; i < 10; i++)
    {
        watch.Restart();
        query(context).ToList();
        watch.Stop();
        totalTicks += watch.ElapsedTicks;
        Console.WriteLine("Compiled: {0}", watch.ElapsedTicks.ToString());
    }
    Console.WriteLine("Average ticks with compiling: {0}",
                       (totalTicks / 10).ToString());
}
```

The following is the output of the code in Listing 13-10:

```
Not Compiled: 4206

Not Compiled: 3359

Not Compiled: 2970

Not Compiled: 2955

Not Compiled: 2899

Not Compiled: 3002

Not Compiled: 2900

Not Compiled: 3042

Not Compiled: 2897

Not Compiled: 2900

Average ticks without compiling: 3113
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Compiled: 4305

Compiled: 511

Compiled: 447

Compiled: 444

Compiled: 461

Compiled: 471

Compiled: 477

Compiled: 443

Compiled: 449

Compiled: 438

Average ticks with compiling: 844
```

How It Works

When you execute a LINQ query, the corresponding expression tree for the query must be converted, or compiled, into an internal command tree. This internal command tree is passed to the provider to be converted into the appropriate database commands (typically SQL). The cost of converting an expression tree can be relatively expensive depending on the complexity of the query and the underlying model. Models with deep inheritance, horizontal splitting, QueryViews, and DefiningQuerys introduce enough complexity in the conversion process that the compile time may become significant relative to the actual query execution time. If the query is reused, you may get a performance improvement by explicitly compiling the query, as illustrated in Listing 13-10.

A compiled query can't contain eSQL or access any builder methods because an eSQL query, by default, is cached. You can turn off caching in eSQL by setting `ObjectQuery.EnablePlanCaching` to `false`. This does not, however, allow you to compile eSQL queries. Only LINQ queries can be compiled.

Compiled queries are especially helpful in ASP.NET search page scenarios where parameter values may change, but the query is the same and can be reused on each page rendering. This works because a compiled query is independent of the object context. The object context for the query is a parameter when executing a compiled query.

A compiled query can return a single entity or a collection of entities, a complex type or a primitive type. If you want to return an anonymous type, you'll need to use the nongeneric version that infers the type from the query itself. This is illustrated in Listing 13-11.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

*Listing 13-11. Using **Compile()** for queries that return an anonymous type*

```
using (var context = new EFRecipesEntities())
{
    // an example of returning an anonymous type
    var list = CompiledQuery.Compile((EFRecipesEntities ctx) =>
        from a in ctx.Associates
        select new
            {
                Name = a.Name,
                TotalPay = a.Paychecks.Sum(p => p.Gross)
            });
    var everyone = list(context).ToList();
}
```

In some cases, you may find that only part of a query is reused in an application. Because compiled queries can be composed with other queries, even in these cases we can often get a performance improvement. The code in Listing 13-12 illustrates using a compiled query in another query.

Listing 13-12. Composing compiled and noncompiled queries

```
using (var context = new EFRecipesEntities())
{
    // an example of composing compiled and non-compiled queries
    var query = CompiledQuery.Compile<EFRecipesEntities, string,
                  IQueryable<Associate>>((ctx, city) =>
                        ctx.Associates.Where(c => c.City == city));
    var highlyPaid = from a in query(context,"Raytown")
                     where a.Paychecks.Average(p => p.Gross) > 1500M
                     select a;
    var associates = highlyPaid.ToList();
}
```

13-8. Returning Partially Filled Entities

Problem

You have a property on an entity that is seldom read and updated. This property is expensive to read and update because of its size. To improve performance, you want to selectively populate this property.

Solution

Let's say you have a model like the one in Figure 13-8.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

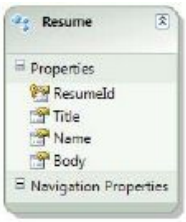


Figure 13-8. *A model with a Resume entity with a Body property that contains the entire text of the applicant's resume*

There are two simple ways to avoid loading one or more properties on an entity. We can partially fill the entity using the constructor in eSQL, or we can use the **ExecuteStoreQuery()** method on the object context to execute a SQL statement. The code in Listing 13-13 illustrates both approaches.

Listing 13-13. Returning partially filled entities using both eSQL and ***ExecuteStoreQuery()***

```
using (var context = new EFRecipesEntities())
{
    var r1 = new Resume { Title = "C# Developer", Name = "Sally Jones",
                          Body = "...very long resume goes here..." };
    context.Resumes.AddObject(r1);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // using the constructor
    var query = @"select value Recipe8.Resume(r.ResumeId, r.Title, r.Name,'')
                  from Resumes as r";
    var result1 = context.CreateQuery<Resume>(query).Single();
    Console.WriteLine("Resume body: {0}", result1.Body);

    context.Resumes.MergeOption = MergeOption.OverwriteChanges;
    var result2 = context.Resumes.Single();
    Console.WriteLine("Resume body: {0}", result2.Body);
}

using (var context = new EFRecipesEntities())
{
    // using ExecuteStoreQuery()
    var result1 = context.ExecuteStoreQuery<Resume>(@"select ResumeId, Title,
                    Name,'' Body from chapter13.Resume", "Resumes",
                    MergeOption.AppendOnly).Single();
    Console.WriteLine("Resume body: {0}", result1.Body);

    var result2 = context.ExecuteStoreQuery<Resume>(@"select * from
                      chapter13.Resume", "Resumes",
                      MergeOption.OverwriteChanges).Single();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    Console.WriteLine("Resume body: {0}", result2.Body);
}
```

The following is the output of the code in Listing 13-13:

```
Resume body:

Resume body: ...very long resume goes here...

Resume body:

Resume body: ...very long resume goes here...
```

How It Works

In the first query in Listing 13-13, we use the constructor for the Resume entity and selectively fill all the properties except for the Body property which we initialize to the empty string. If we later need to populate the Body property from the database, we simply change the `MergeOption` to `MergeOption.OverwriteChanges` and requery the database. The entire entity, including the Body property, is refreshed from the database. This, of course, will overwrite any changes we've made to the object in memory.

Another approach for partially filling an entity is to use the `ExecuteStoreQuery()` method on the object context. Here we execute a SQL statement that fills all the properties except for the Body property, which we initialize to the empty string. As with the first method, we can fill in the Body property from the database by setting the `MergeOption` to `MergeOption.OverwriteChanges` and requerying for the object for all the properties.

The following recipe shows a model-centric and perhaps cleaner approach for this problem.

13-9. Moving an Expensive Property to Another Entity

Problem

You want to move a property to another entity so that you can lazy load that entity. This is often helpful if the property is particularly expensive to load and rarely used.

Solution

As with the previous recipe, let's say you have a model that looks like the one in Figure 13-9.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

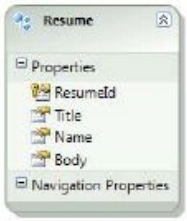


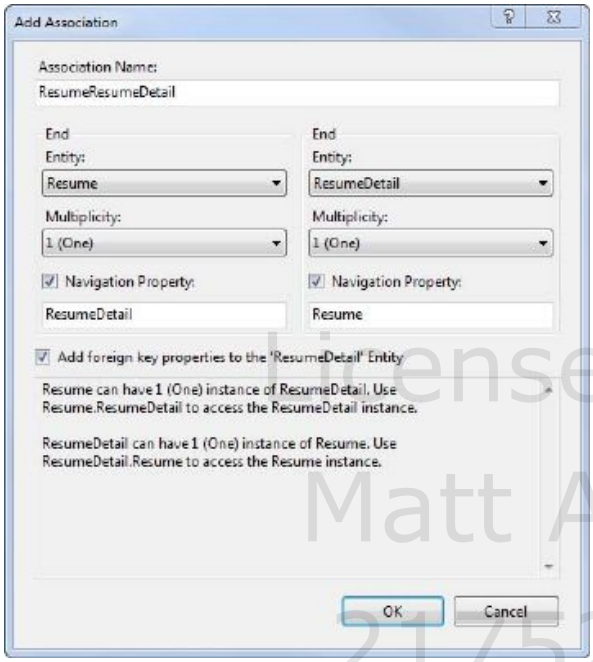
Figure 13-9. *A model with a Resume entity with a Body property that contains the entire text of the applicant's resume. In this recipe, we'll move the Body property to another entity.*

We'll assume, as we did in the previous recipe, that the Body property for the Resume may contain a rather large representation of the applicant's resume. We want to move this property to another entity so that we can lazy load only if we really want to read the resume.

To move the Body property to another entity, do the following:

1. Right-click the design surface and select Add ➤ Entity. Name the new entity **ResumeDetail** and uncheck the Create key property check box.
2. Move the Body property from the Resume entity to the ResumeDetail entity. You can use Cut/Paste to move the property.
3. Right-click the design surface and select Add ➤ Association. Set the multiplicity to One on the Resume side and One on the ResumeDetail side. Check the Add foreign key properties box. See Figure 13-10.
4. Change the name of the foreign key that was created by the association from ResumeResumeId to just **ResumeId**.
5. Select the ResumeDetail entity and view the Mapping Details window. Map the entity to the Resume table. Map the Body property to the Body column. Map the ResumeId property to the ResumeId column. See Figure 13-11.
6. Select the ResumeId property on the ResumeDetail entity and view the properties. Change the EntityKey property to `true`. This marks the ResumeId property as the entity's key. The completed model is shown in Figure 13-12.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



Licensed by Matt Allen 21/5280

Figure 13-10. *Adding an association between Resume and ResumeDetail*

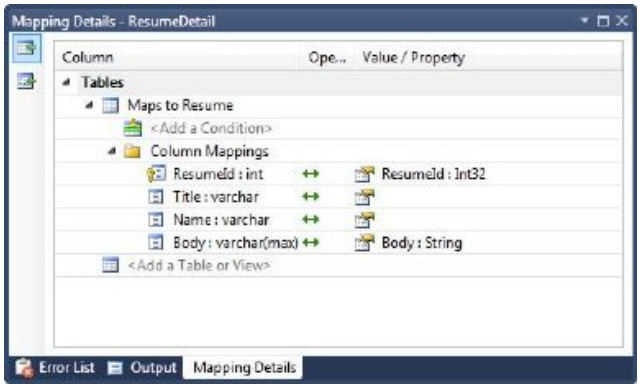


Figure 13-11. *Map the ResumeDetail entity to the Resume table. Map the ResumeId and Body properties as well.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

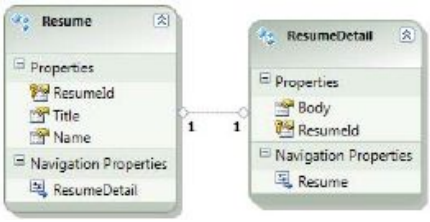


Figure 13-12. *The completed model with the Body property moved to the new ResumeDetail entity*

The code in Listing 13-14 demonstrates how to use the ResumeDetail entity.

Listing 13-14. Using the ResumeDetail entity to lazy load the expensive Body property

```
using (var context = new EFRecipesEntities())
{
    var r1 = new Resume { Title = "C# Developer", Name = "Sally Jones" };
    r1.ResumeDetail = new ResumeDetail {
                          Body = "...very long resume goes here..." };
    context.Resumes.AddObject(r1);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    var resume = context.Resumes.Single();
    Console.WriteLine("Title: {0}, Name: {1}", resume.Title, resume.Name);

    // note, the ResumeDetail is not loaded until we reference it
    Console.WriteLine("Body: {0}", resume.ResumeDetail.Body);
}
```

The following is the output of the code in Listing 13-14:

```
Title: C# Developer, Name: Sally Jones
Body: ...very long resume goes here...
```

How It Works

We avoided loading the expensive Body property on the Resume entity by moving the property to a new related entity. By splitting the underlying table across these two entities, we can exploit the default lazy loading of Entity Framework so that the Body property is loaded only when we reference it. This is a fairly clean approach to the problem, but does introduce an additional entity into our model that we have to manage in our code.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

13-10. Avoiding Include

Problem

You want to eagerly load a related collection without using **Include()**.

Solution

Let's say you have a model like the one in Figure 13-13.

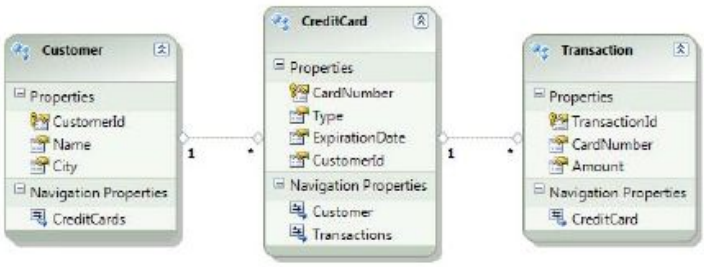


Figure 13-13. *A model for a Customer, their CreditCards and Transactions*

To load all the Customers in a given city together with their credit cards and transactions without using **Include()**, explicitly load the entities and let Entity Framework fix up the associations as shown in Listing 13-15.

Listing 13-15. Loading related entities without using ***Include()***

```
using (var context = new EFRecipesEntities())
{
    var cust1 = new Customer { Name = "Robin Rosen", City = "Raytown" };
    var card1 = new CreditCard { CardNumber = "41949494338899",
                    ExpirationDate = DateTime.Parse("12/2010"), Type = "Visa" };
    var trans1 = new Transaction { Amount = 29.95M };
    card1.Transactions.Add(trans1);
    cust1.CreditCards.Add(card1);
    var cust2 = new Customer { Name = "Bill Meyers", City = "Raytown" };
    var card2 = new CreditCard { CardNumber = "41238389484448",
                    ExpirationDate = DateTime.Parse("12/2013"), Type = "Visa" };
    var trans2 = new Transaction { Amount = 83.39M };
    card2.Transactions.Add(trans2);
    cust2.CreditCards.Add(card2);
    context.Customers.AddObject(cust1);
    context.Customers.AddObject(cust2);
    context.SaveChanges();
}
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
using (var context = new EFRecipesEntities())
{
    var customers = context.Customers.Where(c => c.City == "Raytown");
    var creditCards = customers.SelectMany(c => c.CreditCards);
    var transactions = creditCards.SelectMany(cr => cr.Transactions);

    // execute queries, EF fixes up associations
    customers.ToList();
    creditCards.ToList();
    transactions.ToList();

    foreach (var customer in customers)
    {
        Console.WriteLine("Customer: {0} in {1}", customer.Name, customer.City);
        foreach (var creditCard in customer.CreditCards)
        {
            Console.WriteLine("\tCard: {0} expires on {1}",
                creditCard.CardNumber,
                creditCard.ExpirationDate.ToShortDateString());
            foreach (var trans in creditCard.Transactions)
            {
                Console.WriteLine("\t\tTransaction: {0}",
                    trans.Amount.ToString("C"));
            }
        }
    }
}
```

The following is the output of the code in Listing 13-15:

```
Customer: Robin Rosen in Raytown

        Card: 41949494338899 expires on 12/1/2010

                Transaction: $29.95

Customer: Bill Meyers in Raytown

        Card: 41238389484448 expires on 12/1/2013

                Transaction: $83.39
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

The `Include()` method is a powerful and usually efficient way to eagerly load related entities. However, `Include()` does have some performance drawbacks. Although using `Include()` results in just one round trip to the database in place of the three shown in Listing 13-15, the single query is quite complex and, in some cases, may not perform as well as three much simpler queries. Additionally, the result set from this single, more complex query, contains duplicate columns that increases the amount of data sent over the wire if the database server and the application are on separate machines.

13-11. Improving QueryView Performance

Problem

You have an existing Table per Hierarchy model that is mapped using QueryView. You want to improve the query generation when a specific derived entity is requested.

Solution

Let's say you have a model that uses Table per Hierarchy like the one in Figure 13-14.

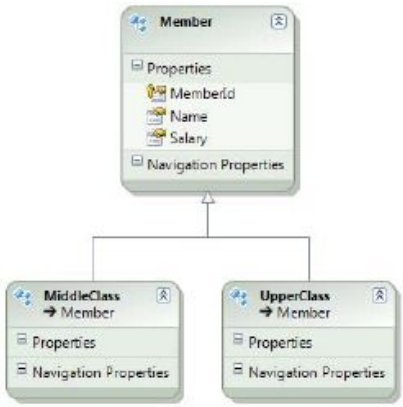


Figure 13-14. *A Table per Hierarchy model for members who are either MiddleClass or UpperClass*

Here we have two derived types: MiddleClass and UpperClass. MiddleClass members make a salary of less than $70,000. UpperClass members make $70,000 and up. We use a QueryView to map these derived types because the `<` and `>` operators are not supported for mapping inheritance.

We might be tempted to use just one QueryView similar to the one in Listing 13-16. However, we can reduce the complexity of the query and gain some performance improvement when we are retrieving a derived type by adding QueryViews for each of the derived types as shown in Listing 13-17.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 13-16. The QueryView mapping the UpperClass and MiddleClass entities

```
<EntitySetMapping Name="Members">
  <QueryView>
    select value
    case when m.Salary &gt; 70000 then
    EFRecipesModel.UpperClass(m.MemberId, m.Name, m.Salary)
    else EFRecipesModel.MiddleClass(m.MemberId, m.Name, m.Salary) end
    from EFRecipesModelStoreContainer.Member as m
  </QueryView>
</EntitySetMapping>
```

Listing 13-17. Improving performance by adding a QueryView for each derived type

```
<EntitySetMapping Name="Members">
  <QueryView>
    select value
    case when m.Salary &gt; 70000 then
    EFRecipesModel.UpperClass(m.MemberId, m.Name, m.Salary)
    else EFRecipesModel.MiddleClass(m.MemberId, m.Name, m.Salary) end
    from EFRecipesModelStoreContainer.Member as m
  </QueryView>
  <QueryView TypeName="IsTypeOf(EFRecipesModel.MiddleClass)">
    select value EFRecipesModel.MiddleClass(m.MemberId, m.Name, m.Salary)
    from EFRecipesModelStoreContainer.Member as m where m.Salary &lt; 70000
  </QueryView>
  <QueryView TypeName="IsTypeOf(EFRecipesModel.UpperClass)">
    select value EFRecipesModel.UpperClass(m.MemberId, m.Name, m.Salary)
    from EFRecipesModelStoreContainer.Member as m where m.Salary &gt; 70000
         or m.Salary = 70000
  </QueryView>
</EntitySetMapping>
```

We use the code in Listing 13-18 to insert a few members into our model and retrieve the UpperClass members.

Listing 13-18. Inserting and retrieving UpperClass members

```
using (var context = new EFRecipesEntities())
{
    context.ExecuteStoreCommand(@"insert into chapter13.member(name,salary)
       values ('Steven Jones',45000)");
    context.ExecuteStoreCommand(@"insert into chapter13.member(name,salary)
       values ('Kathy Kurtz', 85000)");
    context.ExecuteStoreCommand(@"insert into chapter13.member(name,salary)
       values ('Aaron McCabe', 82000)");
}

using (var context = new EFRecipesEntities())
{
    var upperclass = context.Members.OfType<UpperClass>();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var member in upperclass)
    {
        Console.WriteLine("{0}", member.Name);
    }
}
```

The following is the output of the code in Listing 13-18:

```
Kathy Kurtz
Aaron McCabe
```

How It Works

The SQL statement generated by the single QueryView in Listing 13-16 is shown in Listing 13-19. Notice that the **case** statement in our QueryView was translated to a **case** statement in SQL. We can avoid this **case** statement by introducing a QueryView for each of the derived types. This is illustrated in Listing 13-17 and the resulting SQL is shown in Listing 13-20.

Listing 13-19. The case statement from our QueryView makes the generated SQL a little more complex than needed

```
SELECT
CASE WHEN ([Extent1].[Salary] > 70000) THEN '0X0X' ELSE '0X1X' END AS [C1],
[Extent1].[MemberId] AS [MemberId],
[Extent1].[Name] AS [Name],
[Extent1].[Salary] AS [Salary]
FROM [Chapter13].[Member] AS [Extent1]
WHERE CASE WHEN ([Extent1].[Salary] > 70000) THEN '0X0X' ELSE '0X1X' END LIKE '0X0X%'
```

Listing 13-20. A simpler SQL statement tailored to the specific derived type

```
SELECT
'0X0X' AS [C1],
[Extent1].[MemberId] AS [MemberId],
[Extent1].[Name] AS [Name],
[Extent1].[Salary] AS [Salary]
FROM [Chapter13].[Member] AS [Extent1]
WHERE ([Extent1].[Salary] > 70000) OR ([Extent1].[Salary] = 70000)
```

For our simple model, the real runtime differences between Listing 13-19 and Listing 13-20 are rather insignificant. For more complex models, the runtime performance differences can become important.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

13-12. Generating Proxies Explicitly

Problem

You have POCO entities that use dynamic proxies. When you execute a query, you do not want to incur the cost of Entity Framework lazily creating the proxies.

Solution

Suppose you have a model like the one in Figure 13-15.

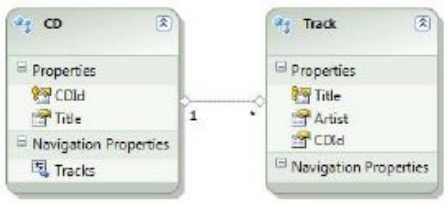


Figure 13-15. *A model for CDs and music titles*

The corresponding POCO classes are shown in Listing 13-21. We have marked each POCO class as **virtual** and set the type for the Tracks property to **ICollection**. This will allow Entity Framework to dynamically create the tracking proxies.

To cause Entity Framework to generate the proxies before they are required (before an entity is loaded), we need to use the **CreateProxyTypes()** method on the object context as illustrated in Listing 13-22.

Listing 13-21. The POCO classes along with our object context

```
public class CD
{
    public virtual int CDId { get; set; }
    public virtual string Title { get; set; }
    public virtual ICollection<Track> Tracks { get; set; }
}

public class Track
{
    public virtual string Title { get; set; }
    public virtual string Artist { get; set; }
    public virtual int CDId { get; set; }
}

public class EFRecipesEntities : ObjectContext
{
    private ObjectSet<CD> _cds;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    private ObjectSet<Track> _tracks;

    public EFRecipesEntities()
        : base("name=EFRecipesEntities", "EFRecipesEntities")
    {
        _cds = CreateObjectSet<CD>();
        _tracks = CreateObjectSet<Track>();
    }

    public ObjectSet<CD> CDs
    {
        get { return _cds; }
    }

    public ObjectSet<Track> Tracks
    {
        get { return _tracks; }
    }
}
```

Listing 13-22. Generating the tracking proxies before loading the entities

```
using (var context = new EFRecipesEntities())
{
    var cd1 = context.CreateObject<CD>();
    cd1.Title = "Abbey Road";
    cd1.Tracks.Add(new Track { Title = "Come Together",
                               Artist = "The Beatles" });
    var cd2 = context.CreateObject<CD>();
    cd2.Title = "Cowboy Town";
    cd2.Tracks.Add(new Track { Title = "Cowgirls Don't Cry",
                               Artist = "Brooks & Dunn" });
    var cd3 = context.CreateObject<CD>();
    cd3.Title = "Long Black Train";
    cd3.Tracks.Add(new Track { Title = "In My Dreams",
                               Artist = "Josh Turner" });
    cd3.Tracks.Add(new Track { Title = "Jacksonville",
                               Artist = "Josh Turner" });
    context.CDs.AddObject(cd1);
    context.CDs.AddObject(cd2);
    context.CDs.AddObject(cd3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // trigger proxy generation
    context.CreateProxyTypes(new Type[] { typeof(CD), typeof(Track) });
    Console.WriteLine("{0} proxies generated!",
                      EFRecipesEntities.GetKnownProxyTypes().Count());

    var cds = context.CDs.Include("Tracks");
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var cd in cds)
    {
        Console.WriteLine("Album: {0}", cd.Title);
        foreach (var track in cd.Tracks)
        {
            Console.WriteLine("\t{0} by {1}", track.Title, track.Artist);
        }
    }
}
```

The following is the output of the code in Listing 13-22:

```
2 proxies generated!

Album: Abbey Road

        Come Together by The Beatles

Album: Cowboy Town

        Cowgirls Don't Cry by Brooks & Dunn

Album: Long Black Train

        In My Dreams by Josh Turner

        Jacksonville by Josh Turner
```

How It Works

Dynamic proxies are created just before they are needed at runtime. This, of course, means that the overhead of creating the proxy is incurred on the first query. This lazy creation approach works well in most cases. You can generate the proxies before the entities are first loaded by calling the **CreateProxyTypes()** method.

The **CreateProxyTypes()** method takes an array of types and generates the corresponding tracking proxies. Once created, the proxies remain in the AppDomain for the life of the AppDomain. Notice that the lifetime of the proxy is tied to the AppDomain, not the object context. We could dispose of the object context and create another and the proxies would not be disposed. You can retrieve the proxies in the AppDomain with the **GetKnownProxyTypes()** static method on the object context.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

13-13. Preventing the Update of All Columns in Self-Tracking Entities

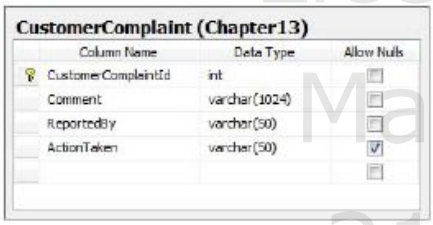
Problem

When using self-tracking entities with Windows Communication Foundation (WCF), you want to make sure that the updates sent contain only columns that have been modified by the client.

Solution

Licensed by Matt Allen 2175280

Suppose you have a table like the one in Figure 13-16 that holds customer complaints.



CustomerComplaint (Chapter13)

Column Name	Data Type	Allow Nulls
CustomerComplaintId	int	☐
Comment	varchar(1024)	☐
ReportedBy	varchar(50)	☐
ActionTaken	varchar(50)	☑
		☐

Figure 13-16. *A table for customer complaints*

When just one property on a self-tracking entity on the client side is modified, the changes on the entity on the service side include all the properties on the entity. The resulting SQL update statement updates all the columns in the underlying table. This can be costly for some entities. You want to update just the columns that changed. To do this, we'll first create a solution complete with separate projects for our entity and context as well as a service project and a test client project. You will likely have all these projects in place already. If so, you can skip to step 12. Otherwise, to create the complete example, do the following:

1. Add a Class Library project to your solution and name it **ComplaintData**. Add an ADO.NET Entity Data Model with the Complaint table.
2. Right-click the design surface and select Add Code Generation Item. Select the ADO.NET Self-Tracking Entity Generator from the Code template. Change the name of the template to **Complaint.tt**. Click Add to add the template to your project.
3. Add a Class Library project to your solution. Call this new project **ComplaintEntities**. Move Complaint.tt from the ComplaintData project to the ComplaintEntities project. Add a reference to System.Runtime.Serialization. Add a project reference in the ComplaintData project to the ComplaintEntities project.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Because we've moved the Complaint.tt template file, we need to edit it to change the reference to the .edmx file for the model. Edit the Complaint.tt template and change the line `string inputFile = @"Recipe13.edmx"` to `string inputFile = @"..\ComplaintData\Recipe13.edmx"`. You may have named your .edmx file something else; if so, make the changes so that the relative path is correct to your .edmx file.

5. Edit the Complaint.Context.tt template (which should still be in the ComplaintData project) and add `using ComplaintEntities;` after each <auto-generated> comment section. This will put the using statement in each generate file.

6. Add a WCF Service Application project to your solution. Name this application **ComplaintService**. Add project references to the ComplaintData and ComplaintEntities. Add a reference to System.Data.Entity.

7. Copy the <connectionStrings> section from the App.Config file in the ComplaintData project to the web.config file in the ComplaintService project. This will allow Entity Framework to connect to the database.

8. Replace the IService1 interface definition IService1.cs file with the code in Listing 13-23.

9. Replace the Service1 implementation in the Service1.svc.cs file with the code in Listing 13-24.

10. Add a Windows Console Application project to the solution. Name the project **TestClient**. The order of the following two steps is important. First, add a reference to the ComplaintEntities project. Next, add a service reference to the ComplaintService. The order is important here because the default behavior (click the Advance button to change this) is to reuse types in all the referenced assemblies. By adding a reference to ComplaintEntities first, when we add the service reference, the CustomerComplaint type from ComplaintEntities is used in place of the ones from ComplaintService.

11. Use the code in Listing 13-25 for the TestClient. Build the solution.

12. Set the TestClient as the Startup Project. Open the SQL Profiler and step through the TestClient. Notice that the call to `UpdateComplaint()` causes the update statement in Listing 13-26 to be sent to the database. This update statement updates all the columns. We only changed the ActionTaken column. To send just the updated column, do the next two steps.

13. Edit the Complaint.tt template file. Change the following lines:

```
OriginalValueMembers originalValueMembers =
  new OriginalValueMembers(allMetadataLoaded, metadataWorkspace, ef);
```

to the following:

```
OriginalValueMembers originalValueMembers =
  new OriginalValueMembers(false, metadataWorkspace, ef);
```

14. Edit the Complaint.Context.tt template file. Change the following line:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
context.ObjectStateManager.ChangeObjectState(entity,
EntityState.Modified);
```

to the following:

```
context.ObjectStateManager.ChangeObjectState(entity,
EntityState.Unchanged);
```

15. Build and run the TestClient again. The SQL Profiler should show that the SQL in Listing 13-27 is sent to the database.

Listing 13-23. The IService1 interface definition in the IService1.cs file

```
using ComplaintEntities;
namespace ComplaintService
{
    [ServiceContract]
    public interface IService1
    {
        [OperationContract]
        void InsertTestRecord();

        [OperationContract]
        CustomerComplaint GetNextComplaint();

        [OperationContract]
        CustomerComplaint UpdateComplaint(CustomerComplaint complaint);
    }
}
```

Listing 13-24. The Service1 implementation in the Service1.svc.cs file

```
using System.Data.Entity;
using ComplaintEntities;
using ComplaintData;
namespace ComplaintService
{
    public class Service1 : IService1
    {
        public void InsertTestRecord()
        {
            using (var context = new EFRecipesEntities())
            {
                context.ExecuteStoreCommand(
                        "delete from chapter13.customercomplaint");
                var complaint = new CustomerComplaint {
                        Comment = "Your store should open early on Saturdays",
                        ReportedBy = "Jill Morgan" };
                context.CustomerComplaints.AddObject(complaint);
                context.SaveChanges();
            }
        }
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        public CustomerComplaint GetNextComplaint()
        {
            using (var context = new EFRecipesEntities())
            {
                var complaint = context.CustomerComplaints
                                .Where(c => c.ActionTaken == null).First();
                complaint.StartTracking();
                return complaint;
            }
        }

        public CustomerComplaint UpdateComplaint(CustomerComplaint complaint)
        {
            using (var context = new EFRecipesEntities())
            {
                context.CustomerComplaints.ApplyChanges(complaint);
                context.SaveChanges();
                complaint.AcceptChanges();
                return complaint;
            }
        }
    }
}
```

Listing 13-25. Code for the TestClient console application. This application is used to test the WCF service.

```
using ComplaintEntities;
using TestClient.ServiceReference1;
namespace TestClient
{
    class Program
    {
        static void Main(string[] args)
        {
            using (var client = new Service1Client())
            {
                // insert a test record
                client.InsertTestRecord();

                var next = client.GetNextComplaint();
                next.ActionTaken = "Your issue is being reviewed";
                client.UpdateComplaint(next);
            }
        }
    }
}
```

Listing 13-26. Original update statement that updates each of the columns

```
exec sp_executesql N'update [Chapter13].[CustomerComplaint]
set [Comment] = @0, [ReportedBy] = @1, [ActionTaken] = @2
  where ([CustomerComplaintId] = @3)',
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
  N'@0 varchar(1024),@1 varchar(50),@2 varchar(50),@3 int',
    @0='Your store should open early on Saturdays',
    @1='Jill Morgan',
    @2='Your issue is being reviewed',
    @3=12
```

Listing 13-27. New update statement that updates only the column that changed

```
exec sp_executesql N'update [Chapter13].[CustomerComplaint]
set [ActionTaken] = @0
  where ([CustomerComplaintId] = @1)',
   N'@0 varchar(50),@1 int',@0='Your issue is being reviewed',@1=18
```

How It Works

In the code generated by the Self-Tracking Entities T4 template, Entity Framework captures the original values of properties that are required in an update statement. These include the primary key and the columns that participate in concurrency. If your model uses independent associations, Entity Framework will keep track of the original foreign key values of the relationships because these are always part of the generated update statements. The end result of this approach is that when you update any property of a self-tracking entity on the client side, Entity Framework simply marks the entity as modified. This results in the update statement updating every column with all the property values, including the properties that did not change.

Why is this the default behavior? The Entity Framework development team found that in most cases sending less information over the wire and updating all columns was faster than sending the additional information about which properties changed. It is faster to send less information and do more work on the database side than send more information and do less work on the database side. Of course, this is true in most cases. If you happen to have a case in which updating all the columns is significantly more expensive than sending the additional information, you should consider making the changes illustrated in this recipe.

With the changes we made to the templates, the self-tracking entity will grab the original values of just the properties that are modified along with the primary key and concurrency columns. When these are played back on the server side, only the changed properties are marked as modified. The resulting SQL update statement updates just the modified columns.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 14

Concurrency

Most applications that use sophisticated database management systems like Microsoft's SQL Server are used by more than one person at a time. The concurrency concerns around shared access to simple data files is often the motivating reason developers turn to relational database systems to support their applications. Many, but not all, of the concurrency concerns evaporate when an application relies on a relational database for its data store. The concerns that remain usually involve detecting and controlling when an object state is different in memory than in the database. The recipes in this chapter provide an introduction to solving some of the problems typically faced by developers when it comes to detecting concurrency violations and controlling which copy of the object is ultimately persisted in the database.

14-1. Applying Optimistic Concurrency

Problem

You want to use optimistic concurrency with an entity in your model.

Solution

Let's suppose you have model like the one shown in Figure 14-1.

Figure 14-1. *A Product entity describing products in your application*

The Product entity describes products in your application. You want to throw an exception if an intermediate update occurs between the time you retrieve a particular product entity and the time an update is performed in the database. To implement that behavior, do the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

1. Add a column of type TimeStamp to the table mapped to the Product entity.
2. Right-click the design surface and select Update Model from Database. Update the model with the newly changed table. The updated model is shown in Figure 14-2.
3. Right-click the TimeStamp property and select Properties. Change its Concurrency Mode property to Fixed.

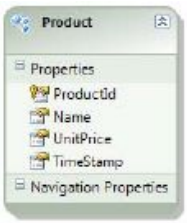


Figure 14-2. *The updated model with the newly added TimeStamp property*

The code in Listing 14-1 demonstrates that changing the underlying row in the table between the time the product entity is materialized and the time we update the table from changes in the product entity throws an exception.

Listing 14-1. Throwing an exception if optimistic concurrency is violated

```
using (var context = new EFRecipesEntities())
{
    context.Products.AddObject(new Product {
          Name = "High Country Backpacking Tent", UnitPrice = 199.95M });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // get the product
    var product = context.Products.SingleOrDefault();
    Console.WriteLine("{0} Unit Price: {1}", product.Name,
                      product.UnitPrice.ToString("C"));

    // update out of band
    context.ExecuteStoreCommand(@"update chapter14.product set
           unitprice = 229.95 where productId = @p0", product.ProductId);

    // update the product the via the model
    product.UnitPrice = 239.95M;
    Console.WriteLine("Changing {0}'s Unit Price to: {1}", product.Name,
                      product.UnitPrice.ToString("C"));

    try
    {
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        context.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        Console.WriteLine("Concurrency Exception! {0}", ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine("Exception! {0}", ex.Message);
    }
}
```

The following is the output of the code in Listing 14-1:

```
High Country Backpacking Tent Unit Price: $199.95

Changing High Country Backpacking Tent's Unit Price to: $239.95

Concurrency Exception! Store update, insert, or delete statement affected an unexpected
number of rows (0). Entities may have been modified or deleted since entities were loaded.
Refresh ObjectStateManager entries.
```

How It Works

Optimistic concurrency is not enabled by default when tables are imported into a model. To enable optimistic concurrency, change the Concurrency Mode property of one of the entity's properties to Fixed. You do not have to use a TimeStamp property as we did in this recipe. You do need to choose a property that you know will be changed in every update to the underlying table. Typically, you would use a column whose value is generated by the database on each update. The TimeStamp column is a good candidate. If you choose another column, be sure to set the StoreGeneratedPattern property to Computed for the corresponding entity property. This will tell Entity Framework that the value is generated by the database. Entity Framework recognizes the TimeStamp data type as a Computed property.

In Listing 14-1, we inserted a new product into the database. We queried the model for the one product we inserted. Once we had the product, we updated the row out-of-band using the **ExecuteStoreCommand()** method to send a SQL **update** statement to the database changing the row. On the database side, this update caused the UnitPrice to be changed to $229.95 and the TimeStamp column to be updated automatically by the database. After the out-of-band update, we changed the UnitPrice on the product in the object context to $239.95. At this point, the object context believes (incorrectly) that it has the most recent values for the product, including an update to the UnitPrice now set at $239.95. When we call **SaveChanges()**, Entity Framework generates an **update** statement with a **where** clause that includes both the ProductId and the TimeStamp values we have for the product. The value for this TimeStamp is one retrieved when we read the product from the database before the out-of-band update. Because the out-of-band update caused the TimeStamp to change, the value for the TimeStamp column in the database is different than the value of the TimeStamp property on the product entity in the object context. The **update** statement will fail because no row is found in the table matching both the ProductId and the TimeStamp values. Entity Framework will respond by rolling back the entire transaction and throwing an OptimisticConcurrencyException.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In responding to the exception, the code in Listing 14-1 printed a message and continued. This is probably not how you would handle a concurrency violation in a real application. One way to handle this exception is to refresh the entity with the current value of the concurrency column from the database. With the correct value for the concurrency column, a subsequent `SaveChanges()` will likely succeed. Of course, it might not for the same reason that it failed the first time, and you need to be prepared for this as well.

The `Refresh()` method on the object context is used to refresh an entity including the current value of the concurrency column. It takes two parameters: a `RefreshMode` and an entity. With `RefreshMode.ClientWins`, property changes made to the entity in the object context are not updated from the database. Only the values, such as the concurrency column value, that are different in the database but not changed in the object context are updated on the entity. In short, changes made in the object context (client) are kept.

With `RefreshMode.StoreWins`, changes made to the entity in the object context are overwritten by values from the database. The database wins.

After calling `Refresh()`, the entity in the object context has the values reflecting either the client winning or the database winning. Calling `SaveChanges()` at this point will generate a new update statement with the new values. This update will either succeed or fail with possibly another OptimisticConcurrencyException if an intervening update statement once again changed the values in the database.

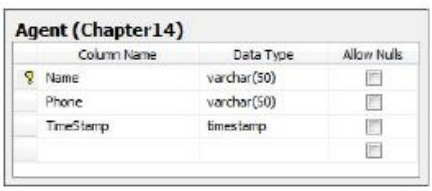
14-2. Managing Concurrency When Using Stored Procedures

Problem

You want to use optimistic concurrency when using stored procedures for the insert, update, and delete actions.

Solution

Let's suppose we have a table like the one shown in Figure 14-3 and the entity shown in Figure 14-4 that is mapped to the table.



Agent (Chapter14)

Column Name	Data Type	Allow Nulls
Name	varchar(50)	☐
Phone	varchar(50)	☐
TimeStamp	timestamp	☐
		☐

Figure 14-3. *The Agent table in our database*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Figure 14-4. *Our model with the Agent entity*

You want to use stored procedures to handle the insert, update, and delete actions for the model. These stored procedures need to be written so that they leverage the optimistic concurrency support provided in Entity Framework. Do the following to create the stored procedures, import them into the model, and map them to actions:

Licensed by Matt Allen 21 45280

1. Create the stored procedures in the database using the code in Listing 14-2.
2. Right-click the design surface and select Update Model from Database. Select the stored procedures you created in step 1. Click Finish. This will import the stored procedures into the model.
3. View the Mapping Details window for the Agent entity. Click the Map Entity to Functions button on the left side of the tool window. Map the insert, update, and delete actions to the stored procedures, as shown in Figure 14-5. Make sure you map the Result column to the TimeStamp property for both the insert and update actions. For the update action, check the Use Original Value check box for the procedure's TimeStamp parameter.

Listing 14-2. Stored procedures for the insert, update, and delete actions

```
create procedure Chapter14.InsertAgent
(@Name varchar(50), @Phone varchar(50))
as
begin
  insert into Chapter14.Agent(Name, Phone) values (@Name, @Phone)
  select TimeStamp from Chapter14.Agent where Name = @Name and @@ROWCOUNT > 0
end
go
create procedure Chapter14.UpdateAgent
(@Name varchar(50), @Phone varchar(50), @TimeStamp TimeStamp)
as
begin
  update Chapter14.Agent set Phone = @Phone where Name = @Name and TimeStamp = @TimeStamp
  select TimeStamp from Chapter14.Agent where Name = @Name and @@ROWCOUNT > 0
end
go
create procedure Chapter14.DeleteAgent
(@Name varchar(50), @TimeStamp TimeStamp)
as
begin
  delete Chapter14.Agent where Name = @Name and TimeStamp = @TimeStamp
end
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

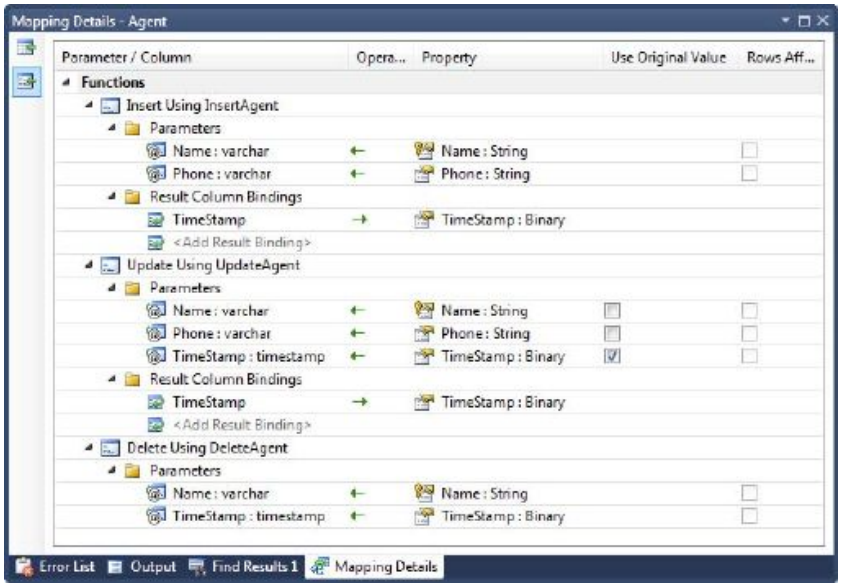


Figure 14-5. *Mapping the insert, update, and delete actions to the stored procedures*

The code in Listing 14-3 demonstrates inserting and updating the database using the stored procedures. In the code, we update the phone numbers for both agents. For the first agent, we update the agent in the object context and save the changes. For the second agent, we do an out-of-band update before we update the phone using the object context. When we save the changes, Entity Framework throws an OptimisticConcurrencyException, indicating that the underlying database row was modified after the agent was materialized in the object context.

Listing 14-3. Demonstrating how Entity Framework and our insert and update stored procedures respond to a concurrency violation

```
using (var context = new EFRecipesEntities())
{
    context.Agents.AddObject(new Agent { Name = "Phillip Marlowe",
                                         Phone = "202 555-1212" });
    context.Agents.AddObject(new Agent { Name = "Janet Rooney",
                                         Phone = "913 876-5309" });
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // change the phone numbers
    var agent1 = context.Agents.Where(a => a.Name == "Janet Rooney").Single();
    var agent2 = context.Agents.Where(a => a.Name == "Phillip Marlowe").Single();
    agent1.Phone = "817 353-4458";
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.SaveChanges();

    // update the other agent's number out-of-band
    context.ExecuteStoreCommand(@"update Chapter14.agent
         set Phone = '817 294-6059' where name = 'Phillip Marlowe'");

    // now change it using the model
    agent2.Phone = "817 906-2212";
    try
    {
        context.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        Console.WriteLine("Exception caught updating phone number: {0}",
                          ex.Message);
    }
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("-- All Agents --");
    foreach (var agent in context.Agents)
    {
        Console.WriteLine("Agent: {0}, Phone: {1}", agent.Name, agent.Phone);
    }
}
```

The following is the output of the code in Listing 14-3. Notice that we caught the exception thrown during **SaveChanges()** and printed the exception message:

```
Exception caught updating phone number: Store update, insert, or delete statement

affected an unexpected number of rows (0). Entities may have been modified or deleted since
entities were loaded. Refresh ObjectStateManager entries.

-- All Agents --

Agent: Janet Rooney, Phone: 817 353-4458

Agent: Phillip Marlowe, Phone: 817 294-6059
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

The key to leveraging the concurrency infrastructure in Entity Framework is in the implementation of the stored procedures (see Listing 14-2) and in how we mapped the input parameters and the result values. Let's look at each stored procedure.

The `InsertAgent()` procedure takes in the name and phone number for the agent and executes an `insert` statement. This results in the database computing a timestamp value that is inserted into the table along with the name and phone number. The `select` statement retrieves this newly generated timestamp. We map this result column (see Figure 14-5) to the TimeStamp property on the entity. After the insert, the entity has the current values for the name and phone number and the newly generated timestamp. At that instant, the entity is in sync with the database.

With the `UpdateAgent()` procedure, we map the Name, Phone, and TimeStamp properties from the entity to the corresponding parameters of the procedure. We checked the Use Original Value check box for the TimeStamp property. This ensures that the original value for the TimeStamp property is sent to the database. The `where` clause on the `update` statement includes the timestamp. If the timestamp value for the row in the database is different from the value in the entity, the update will fail. Because no rows are updated, Entity Framework responds by throwing an OptimisticConcurrencyException. If the update succeeds, the new timestamp value is mapped to the TimeStamp property on the entity. At this point, the entity and row in the table are once again synchronized.

For the `DeleteAgent()` procedure, we map the Name and TimeStamp properties to the parameters of the procedure. The `where` clause on the delete statement includes the primary key, Name, and the timestamp value. This ensures that the row is deleted if and only if no intermediate update of the row has occurred. If no row is deleted, Entity Framework will respond with an OptimisticConcurrencyException.

Entity Framework relies on each of these stored procedures returning some indication of the number of rows affected by the operation. We've crafted each procedure to return this value either using a select statement that returns either one or zero rows, or in the case of `DeleteAgent()`, the stored procedure returns the row count from the delete statement.

There are three ways, in order of precedence, that Entity Framework interprets the number of rows affected by a stored procedure: the return value from `ExecuteNonQuery()`, the number of rows returned, or an explicit output parameter (see Recipe 14-5).

The code in Listing 14-3 demonstrates that an intervening update, which we do out-of-band with the `ExecuteStoreCommand()` method, causes a concurrency violation when we update Phillip Marlowe's phone number.

14-3. Reading Uncommitted Data

Problem

You want to read uncommitted data using LINQ to entities.

Solution

Suppose you have an Employee entity like the one shown in Figure 14-6. You want to insert a new employee, but before the row is committed to the database, you want to read the uncommitted row into a different object context. To do this, create nested instances of the TransactionScope class and set the

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

IsolationLevel of the innermost scope to **ReadUncommitted** as shown in Listing 14-4. You will need to add a reference in your project to System.Transactions.

Figure 14-6. *An Employee entity*

Listing 14-4. Creating nested TransactionScopes and setting the IsolationLevel to ***ReadUncommitted***

```
using (var context = new EFRecipesEntities())
{
    using (var scope1 = new TransactionScope())
    {
        // save, but don't commit
        var outerEmp = new Employee { Name = "Karen Stanfield" };
        Console.WriteLine("Outer employee: {0}", outerEmp.Name);
        context.Employees.AddObject(outerEmp);
        context.SaveChanges();

        // second transaction for read uncommitted
        using (var innerContext = new EFRecipesEntities())
        {
            using (var scope2 = new TransactionScope(
                TransactionScopeOption.RequiresNew,
                new TransactionOptions {
                   IsolationLevel = IsolationLevel.ReadUncommitted }))
            {
                var innerEmp = innerContext.Employees
                                .First(e => e.Name == "Karen Stanfield");
                Console.WriteLine("Inner employee: {0}", innerEmp.Name);
                scope1.Complete();
                scope2.Complete();
            }
        }
    }
}
```

The following is the output of the code in Listing 14-4:

```
Outer employee: Karen Stanfield
Inner employee: Karen Stanfield
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

In SQL, one of the common ways of reading uncommitted data is to use the NOLOCK query hint. However, Entity Framework does not support the use of hints. In Listing 14-4, we used a TransactionScope with the IsolationLevel set to ReadUncommitted. This allowed us to read the uncommitted data from the outer TransactionScope. We did this in a fresh object context.

14-4. Implementing the "Last Record Wins" Strategy

Problem

You want to make sure that changes to an object succeed regardless of any intermediate changes to the database.

Solution

Suppose you have a model like the one shown in Figure 14-7.

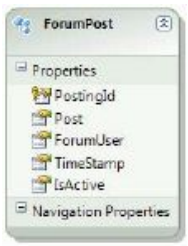


Figure 14-7. *Our model with the ForumPost entity*

Our model represents posts by users of an Internet forum. Moderators of forums often want to review posts and possibly change or delete them. The changes a moderator makes need to take precedence over any changes made by the forum's users. In general, this can be implemented without much concern for concurrency, except when the user makes a change between the time the moderator retrieves the post and the when the moderator calls **SaveChanges()** to commit a change such as a delete to the database. In this case, we want the moderator's changes to overwrite the user's changes. We want the moderator to win.

To implement this, follow the pattern in Listing 14-5. Be sure to set the Concurrency Mode on the TimeStamp property to Fixed.

Listing 14-5. Implementing last record wins

```
int postId = 0;
using (var context = new EFRecipesEntities())
{
    // post is created
    var post = new ForumPost { ForumUser = "FastEddie27", IsActive = false,
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                Post = "The moderator is a great guy." };
    context.ForumPosts.AddObject(post);
    context.SaveChanges();
    postId = post.PostingId;
}

using (var context = new EFRecipesEntities())
{
    // moderator gets post to review
    var post = context.ForumPosts.First(p => p.PostingId  == postId);
    Console.WriteLine("Post by {0}: {1}", post.ForumUser, post.Post);

    // poster changes post out-of-band
    context.ExecuteStoreCommand(@"update chapter14.forumpost
             set post='The moderator''s mom dresses him funny.'
             where postingId = @p0", new object[] { postId.ToString() });
    Console.WriteLine("Fast Eddie changes the post");

    // moderator doesn't trust Fast Eddie
    if (string.Compare(post.ForumUser, "FastEddie27") == 0)
        post.IsActive = false;
    else
        post.IsActive = true;

    try
    {
        // refresh any changes to the TimeStamp
        context.ForumPosts.MergeOption = MergeOption.PreserveChanges;
        post = context.ForumPosts.First(p => p.PostingId == postId);
        context.SaveChanges();
        Console.WriteLine("No concurrency exception.");
    }
    catch (OptimisticConcurrencyException)
    {
        try
        {
            context.Refresh(RefreshMode.ClientWins, post);
            context.SaveChanges();
        }
        catch (OptimisticConcurrencyException)
        {
            // we tried twice...do something else
        }
    }
}
```

The following is the output of the code in Listing 14-5:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Post by FastEddie27: The moderator is a great guy.

Fast Eddie changes the post

No concurrency exception.
```

How It Works

The TimeStamp property is marked for concurrency because its ConcurrencyMode is set to Fixed. As part of the `update` statement, the value of the TimeStamp property is checked against the value in the database. If they differ, Entity Framework will throw an OptimisticConcurrencyException. We've seen this behavior in the previous recipes in this chapter. What's different here is that we want the change from the client, in this case the moderator, to overwrite the newer row in the database. We do this by using two slightly different strategies. Even with these two strategies, there is some chance that our object can't be committed to the database.

The first strategy we use in Listing 14-5 is to change the `MergeOption` for the ForumPosts entity set to `PerserveChanges` and then requery for the object before calling `SaveChanges()`. With the `PerserveChanges` set, the requery will reload the object overwriting only these values that we didn't change in the object context. Most importantly, this means that the TimeStamp property will be refreshed from the database. Armed with the latest TimeStamp property, our call to `SaveChanges()` should succeed. There is a chance, however, especially in a highly concurrent environment, that some intervening update could occur to change the row before our update hits the database. If this occurs, Entity Framework will throw an OptimisticConcurrencyException. We handle this case using our second strategy.

Our second strategy uses the `Refresh()` method on the object context. The `Refresh()` method reloads the object. The first parameter to `Refresh()` is an enum that determines whether the client values (`ClientWins`) or the server values (`ServerWins`) become the final values in the object. In our case, we use `RefreshMode.ClientWins`, which causes the values changed in the object in the object context to remain while the values not changed in the object context are refreshed from the server. Most importantly, this includes the TimeStamp value from the server.

Using the `Refresh()` method is subtly different from the requery we used in our first strategy. `Refresh()` works on a per-entity basis. Every call to `Refresh()` results in a round trip to the database. Our requery strategy works on a per-query basis. All the entities that result from the query are refreshed with the active `MergeOption`.

Even with these two approaches in place, it is still possible for an intervening update to occur between the `Refresh()` or the requery and the time the update is executed on the database.

14-5. Getting Affected Rows from a Stored Procedure

Problem

You want to return the number of rows affected by a stored procedure through an output parameter.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

The Entity Framework uses the number of rows affected by an operation to determine whether the operation succeeded or the operation failed because of a concurrency violation. When using stored procedures (see Recipe 14-2 in this chapter), one of the ways to communicate the number of rows affected by an operation is to return this value as an output parameter of the stored procedure.

Let's suppose you have a model like the one shown in Figure 14-8.

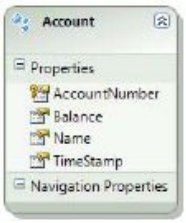


Figure 14-8. *Our model with the Account entity*

To return the number of rows affected by the stored procedures mapped to the insert, update, and delete actions, do the following:

1. Create the stored procedures in the database using the code in Listing 14-6.
2. Right-click the design surface and select Update Model from Database. Select the stored procedures you created in Step 1. Click Finish. This will import the stored procedures into the model.
3. View the Mapping Details window for the Account entity. Click the Map Entity to Functions button on the left side of the tool window. Map the insert, update, and delete actions to the stored procedures as shown in Figure 14-9. Make sure you map the Result column to the TimeStamp property for both the insert and update actions. For the update action, check the Use Original Value box for the procedure's TimeStamp parameter. For each procedure, check the Rows Affected Parameter boxes as shown in Figure 14-9.

Listing 14-6. The stored procedures for the insert, update, and delete actions

```
create procedure [Chapter14].[UpdateAccount]
(@AccountNumber varchar(50), @Name varchar(50), @Balance decimal, @TimeStamp TimeStamp,
@RowsAffected int output)
as
begin
  update Chapter14.Account set Name = @Name, Balance = @Balance
        where AccountNumber = @AccountNumber and TimeStamp = @TimeStamp
  set @RowsAffected = @@ROWCOUNT
  select TimeStamp from Chapter14.Account where AccountNumber = @AccountNumber
end

go
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
create procedure [Chapter14].[InsertAccount]
(@AccountNumber varchar(50), @Name varchar(50), @Balance decimal,
 @RowsAffected int output)
as
begin
  insert into Chapter14.Account (AccountNumber, Name, Balance) values (@AccountNumber, @Name,
@Balance)
  set @RowsAffected = @@ROWCOUNT
  select TimeStamp from Chapter14.Account where AccountNumber = @AccountNumber
end

go

create procedure [Chapter14].[DeleteAccount]
(@AccountNumber varchar(50), @TimeStamp TimeStamp, @RowsAffected int output)
as
begin
  delete Chapter14.Account where AccountNumber = @AccountNumber and
         TimeStamp = @TimeStamp
  set @RowsAffected = @@ROWCOUNT
end
```

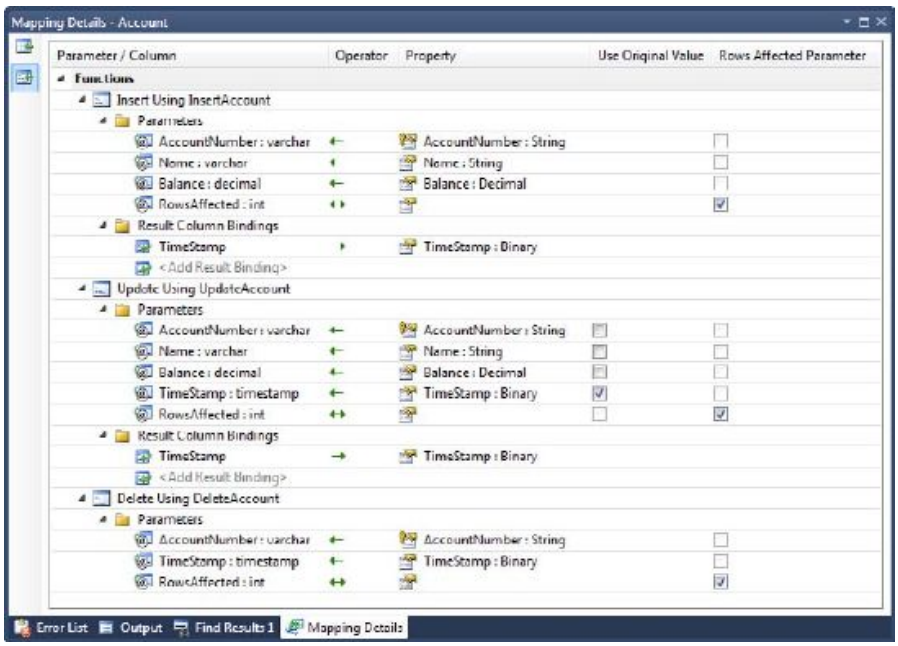


Figure 14-9. *When mapping the stored procedures to the insert, update, and delete actions, make sure you check the Rows Affected Parameter check boxes and Use Original Value check box as shown.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

When we call the **SaveChanges()** method in Listing 14-7 to update, insert, or delete, these actions are performed by the stored procedures in Listing 14-6 because of the mappings in Figure 14-9. Both the insert and update procedures return the updated TimeStamp value. This value is used by Entity Framework to enforce optimistic concurrency.

Listing 14-7. Demonstrating the stored procedures mapped to the insert, update, and delete actions

```
using (var context = new EFRecipesEntities())
{
    context.Accounts.AddObject(new Account { AccountNumber = "8675309",
                               Balance = 100M, Name = "Robin Rosen"});
    context.Accounts.AddObject(new Account { AccountNumber = "8535937",
                               Balance = 25M, Name = "Steven Bishop"});
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // get the account
    var account = context.Accounts.First(a => a.AccountNumber == "8675309");
    Console.WriteLine("Account for {0}", account.Name);
    Console.WriteLine("\tPrevious Balance: {0}", account.Balance.ToString("C"));

    // some other process updates the balance
    Console.WriteLine("[Rogue process updates balance!]");
    context.ExecuteStoreCommand(@"update chapter14.account set balance = 1000
                                  where accountnumber = '8675309'");

    // update the account balance
    account.Balance = 10M;

    try
    {
        Console.WriteLine("\tNew Balance: {0}", account.Balance.ToString("C"));
        context.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        Console.WriteLine("Exception: {0}", ex.Message);
    }
}
```

The following is the output of the code in Listing 14-7:

```
Account for Robin Rosen

        Previous Balance: $100.00

[Rogue process updates balance!]
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        New Balance: $10.00

Exception: Store update, insert, or delete statement affected an unexpected number of rows
(0). Entities may have been modified or deleted since entities were loaded. Refresh
ObjectStateManager entries.
```

How It Works

The code in Listing 14-7 demonstrates using the stored procedures we've mapped to the insert, update, and delete actions. In the code, we purposely introduce an intervening update between the retrieval of an account object and saving the account object to the database. This rogue update causes the TimeStamp value to be changed in the database after we've materialized the object in the object context. This concurrency violation is detected by Entity Framework because the number of rows affected by the `UpdateAccount()` procedure is zero.

The mappings shown in Figure 14-9 tell Entity Framework how to keep the TimeStamp property correctly synchronized with the database and how to be informed of the number of rows affected by the insert, update, or delete actions. The Result Column for the insert and the update actions is mapped to the TimeStamp property on the entity. For the update action, we need to make sure that Entity Framework uses the original value from the entity when it constructs the statement invoking the `UpdateAccount()` procedure. These two settings keep the TimeStamp property synchronized with the database. Because our stored procedures return the number of rows affected the actions in an output parameter, we need to check the Rows Affected Parameter box for this parameter for each of the action mappings.

14-6. Optimistic Concurrency with Table Per Type Inheritance

Problem

You want to use optimistic concurrency in a model that uses Table per Type inheritance.

Solution

Let's suppose you have the tables shown in Figure 14-10 and you want to model these tables using Table per Type inheritance and use optimistic concurrency to ensure that updates are persisted correctly. To create the model supporting optimistic concurrency, do the following:

1. Add a TimeStamp column to the Person table.
2. Import the Person, Instructor, and Student tables to your existing model or create a new ADO.NET model with these tables.
3. Remove the associations between the Person and Student entities and between the Person and Instructor entities.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Right-click the design surface and select Add New ➤ Inheritance. Select Person as the base entity and Instructor as the derived entity.
5. Right-click the design surface and select Add New ➤ Inheritance. Select Person as the base entity and Student as the derived entity.
6. Remove the PersonId properties from both the Student and the Instructor entities.
7. Right-click the TimeStamp property in the Person entity and view the Properties. Set the Concurrency Mode to Fixed.

The resulting model is shown in Figure 14-11. The code in Listing 14-8 demonstrates what happens in the model when an out-of-band update happens.

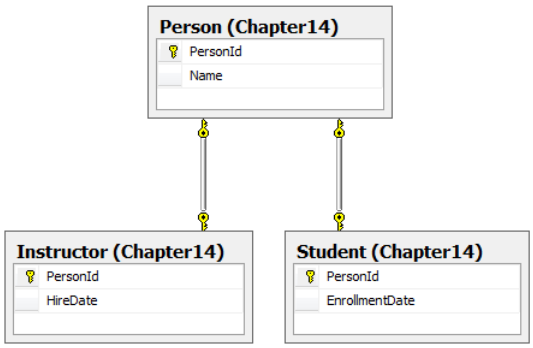


Figure 14-10. *A database diagram with our Person table and the related Instructor and Student tables*

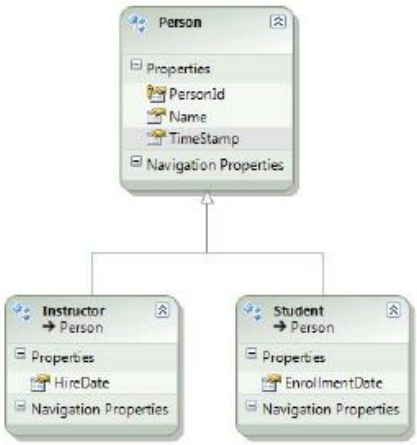


Figure 14-11. *The final model with the derived types Instructor and Student*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 14-8. Testing the model by applying a rogue update

```
using (var context = new EFRecipesEntities())
{
    var student = new Student { Name = "Joan Williams",
                                EnrollmentDate = DateTime.Parse("1/12/2010") };
    var instructor = new Instructor { Name = "Rodger Keller",
                                HireDate = DateTime.Parse("7/14/1992") };
    context.People.AddObject(student);
    context.People.AddObject(instructor);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    // find the student and update the enrollment date
    var student = context.People.OfType<Student>()
                    .First(s => s.Name == "Joan Williams");
    Console.WriteLine("Updating {0}'s enrollment date", student.Name);

    // out-of-band update occurs
    Console.WriteLine("[Apply rogue update]");
    context.ExecuteStoreCommand(@"update chapter14.person set name = 'Joan Smith'
          where personId =
          (select personId from chapter14.person where name = 'Joan Williams')");

    // change the enrollment date
    student.EnrollmentDate = DateTime.Parse("5/2/2010");
    try
    {
        context.SaveChanges();
    }
    catch (OptimisticConcurrencyException ex)
    {
        Console.WriteLine("Exception: {0}", ex.Message);
    }
}
```

The following is the output of the code in Listing 14-8:

```
Updating Joan Williams's enrollment date

[Apply rogue update]

Exception: Store update, insert, or delete statement affected an unexpected number of rows
(0). Entities may have been modified or deleted since entities were loaded. Refresh
ObjectStateManager entries.
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

In Listing 14-8, the code retrieves a student entity. An intervening update occurs to the Person table before the code updates the EnrollmentDate property on the entity and calls **SaveChanges()**. Entity Framework detects the concurrency violation when updating the tables in the database because the value in the TimeStamp column in the Person table does not match the TimeStamp value in the student entity. Entity Framework applies concurrency at the entity level. Before the Student table is updated, the Person table is updated with a meaningless or dummy update and the new TimeStamp value is obtained. This can be seen in the trace in Listing 14-9. If this update fails to affect any rows, Entity Framework knows that the underlying table was changed since the last read. This would cause Entity Framework to throw an OptimisticConcurrencyException.

Listing 14-9. Entity Framework updates the TimeStamp in the base table prior to performing the update in the derived table

```
exec sp_executesql N'declare @p int
update [Chapter14].[Person]
set @p = 0
where (([PersonId] = @0) and ([TimeStamp] = @1))
select [TimeStamp]
from [Chapter14].[Person]
where @@ROWCOUNT > 0 and
[PersonId] = @0',N'@0 int,@1 binary(8)',@0=10,@1=0x0000000000007D19
```

However, if the rogue update occurred on the Student table in the database, the TimeStamp column in the Person table would not have been changed and Entity Framework would not have detected a concurrency violation. This is an important point to remember. The concurrency detection illustrated here extends just to rogue updates to the base entity.

14-7. Generating a Timestamp Column with Model First

Problem

You want to use Model First and you want an entity to have a TimeStamp property for use in optimistic concurrency.

Solution

To use Model First and create an entity with a TimeStamp property, do the following:

1. Find the T4 Template that is used to generate the DDL for Model First. This file is located in Program Files\Microsoft Visual Studio 10.0\Common7\IDE\Extensions\Microsoft\Entity Framework Tools\DBGen\SSDLToSQL10.tt. Copy this file and rename this copy to **SSDLToSQL10Recipe7.tt**. Place the copy in the same folder as the original.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

2. Replace the line that starts with **[<#=Id(prop.Name)#>]** with the code in Listing 14-10. We'll use this modified T4 Template to generate the DDL for our database.
3. Add a new ADO.NET Entity Data Model to your project. Start with an Empty Model.
4. Right-click the design surface and select Add ➤ Entity. Name this new entity **PhonePlan**. Change the Key Property name to **PhonePlanId**. Click OK.
5. Add Scalar properties for Minutes, Cost, and TimeStamp. Change the type for the Minutes property to Int32. Change the type for the Cost property to Decimal.
6. Change the type of the TimeStamp property to Binary. Change its StoreGeneratedPattern to Computed. Change the Concurrency Mode to Fixed.
7. Right-click the design surface and view the Properties. Change the DDL Generation Template to SSDLToSQL10Recipe7.tt. This is the template that you modified in step 2. Change the Database Schema Name to **Chapter14**.
8. Right-click the design surface and click Generate Database from Model. Select the connection and click Next. The generated DDL is shown in the dialog box. Listing 14-11 shows an extract from the generated DDL that creates the PhonePlan table. Click Finish to complete the generation.

Listing 14-10. Replace the line in the the T4 Template with this line.

```
 [<#=Id(prop.Name)#>]
<#if (string.Compare(prop.Name,"TimeStamp",true) == 0)
{#>TIMESTAMP<#} else { #><#=prop.ToStoreType()#><#} #>
<#=WriteIdentity(prop, targetVersion)#> <#=WriteNullable(prop.Nullable)#>
<#=(p < entitySet.ElementType.Properties.Count - 1) ? "," : ""#>
```

Listing 14-11. The DDL that creates the PhonePlan table

```
-- Creating table 'PhonePlans'
CREATE TABLE [Chapter14].[PhonePlans] (
[PhonePlanId] int  IDENTITY(1,1) NOT NULL,
[Minutes] int   NOT NULL,
[Cost] decimal(18,0)   NOT NULL,
[TimeStamp] TIMESTAMP   NOT NULL
);
GO
```

How It Works

The TimeStamp data type is not a portable type. Not all database vendors support it. It is unlikely that this type will be supported at the conceptual layer in future versions of Entity Framework. However, future versions will likely improve the user experience in selecting or modifying the appropriate T4 template that will generate the DDL.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

CHAPTER 15

Advanced Modeling

The Entity Framework runtime supports a surprisingly wide variety of modeling scenarios, even if the current implementation of the designer does have some limitations. In this chapter, we explore some of the more advanced modeling concepts that are supported by Entity Framework. Many of these models represent real-world challenges dealing with existing databases that may not have been designed with the best approaches.

15-1. Creating an Association on a Derived Entity

Problem

You have a table with a nullable foreign key column. You want to model the table using Table per Hierarchy inheritance and map the foreign key column as an association on the derived entity.

Solution

Let's say you have a couple of tables like the ones shown in the diagram in Figure 15-1.

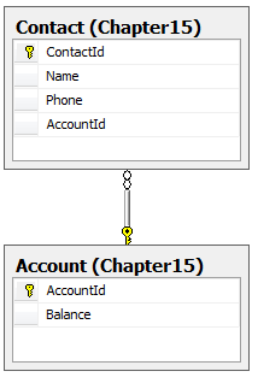


Figure 15-1. *A Contact table with an optional (nullable) foreign key to an Account table*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

In Figure 15-1, we have a Contact table that contains a name and phone number as well as an optional foreign key to related account information held in an Account table. In our model, if a contact has related account information, the contact is a customer.

We want to represent the customer as a derived type in our model. Additionally, we want to represent the related account information as an entity and create an association between the derived type and the account entity.

To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Contact and Account tables. Or update an existing model with these tables.
2. Remove the association created by the designer between the Contact and Account entities.
3. Right-click the design surface and select Add ➤ Entity. Name the new entity **Customer**. Select Contact as the base type for the new Customer entity.
4. Move the AccountId property from the Contact entity to the Customer entity. You can use Cut/Paste to move this property.
5. Select the Customer entity. In the Mapping Details window, select the Contact table in Add a Table or View. Add the condition when `AccountId is not null.`
6. Right-click the design surface and select Add ➤ Association. Create a one-to-many association between the Account entity and the Customer entity. Make sure the Customer is on the many side of the association.
7. Right-click the association and select Properties. Click the Referential Constraint box. In the dialog box, select the Account entity for the Principal role. Select AccountId as the Dependent Property.

The completed model should look like the one in Figure 15-2.

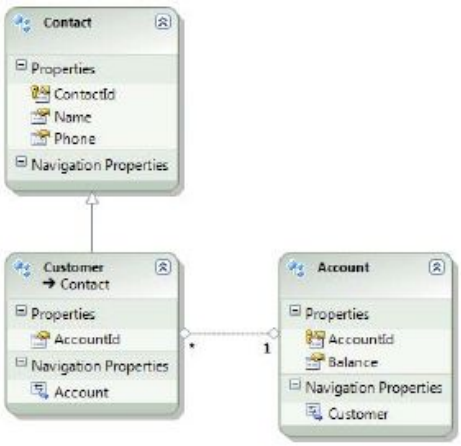


Figure 15-2. *The completed model with a one-to-many association between the Account entity and the derived entity, Customer*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

We used the AccountId column as the discriminator in the Table per Hierarchy inheritance model. We also used the AccountId column in a conditional association to the Account entity.

It may seem strange for the association between Account and Customer to be one-to-many. We know that each customer has exactly one account and that AccountIds are unique across instances of the Contact entity. However, the AccountId foreign key in Contact is not an entity key. From the model's perspective, there is no guarantee that the AccountId foreign key provides for the expected one-to-one relationship to the Account table.

The code in Listing 15-1 demonstrates inserting into and querying our model.

Listing 15-1. Inserting and querying our model

```
using (var context = new EFRecipesEntities())
{
    var acc1 = new Account { Balance = 99.34M };
    var con1 = new Contact { Name = "Stacy Jones", Phone = "867-5301" };
    var cus1 = new Customer { Name = "Bill Waters", Phone = "907-2212",
                              Account = acc1 };
    context.Contacts.AddObject(con1);
    context.Contacts.AddObject(cus1);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    Console.WriteLine("All Contacts");
    Console.WriteLine("============");
    foreach (var contact in context.Contacts)
    {
        Console.WriteLine("{0} {1}", contact.Name, contact.Phone);
    }

    Console.WriteLine("Just Customers");
    foreach (var contact in context.Contacts.OfType<Customer>())
    {
        Console.WriteLine("\t{0} {1} (Balance: {2})", contact.Name,
                          contact.Phone,
                          contact.Account.Balance.ToString("C"));
    }
}
```

The following is the output of the code in Listing 15-1:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
All Contacts

============

Stacy Jones 867-5301

Bill Waters 907-2212

Just Customers

        Bill Waters 907-2212 (Balance: $99.34)
```

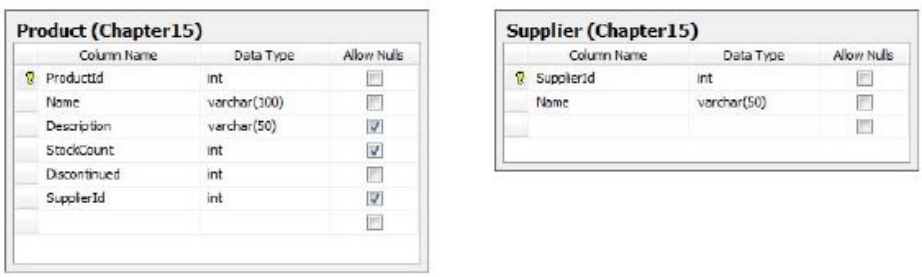
15-2. Mapping an Entity to Customized Parts of One or More Tables

Problem

You want to map an entity to parts of one or more tables, excluding some columns, changing the data types of others and adding a computed column.

Solution

Let's suppose you have the two tables shown in Figure 15-3.



Product (Chapter15)

Column Name	Data Type	Allow Nulls
ProductId	int	☐
Name	varchar(100)	☐
Description	varchar(50)	☑
StockCount	int	☑
Discontinued	int	☐
SupplierId	int	☑
		☐

Supplier (Chapter15)

Column Name	Data Type	Allow Nulls
SupplierId	int	☐
Name	varchar(50)	☐
		☐

Figure 15-3. *Product and Supplier tables*

In Figure 15-3, we have two tables: Product and Supplier. We want to map these tables to a single Product entity with the following changes.

The Name property should be limited to 50 characters rather than the 100 in the underlying Product table. The Description column is not needed. The StockCount property should not be null. If the underlying StockCount column is null, the entity's StockCount property should have a value of 0. The Discontinued column should be a bit field in the entity rather than the integer field in the underlying

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

table. Because you plan to use the supplier's name often, you want to include it as a property in the entity. Finally, you want to expose a computed property called IsBackOrderable on the entity.

To create a model with the Product entity having the correct properties, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Product and Supplier tables. Or update an existing model with these tables.
2. Select the Name property in the Product entity and view its properties. Change its Max Length from 100 to 50.
3. Select the StockCount property and view its properties. Change its Nullable value from True to False.
4. Delete the Description property from the entity.
5. Delete the SupplierId property from the entity.
6. Right-click the Product entity and select Add ➤ Scalar Property. Name the new property **IsBackOrderable**. Set its Type to Boolean. Set its Nullable property to False.
7. Select the Discontinued property and view its properties. Change its Type from Int32 to Boolean.
8. Right-click the Product entity and select Add ➤ Scalar Property. Name the new property **SupplierName** and set its Type to String with a Max Length of 50.
9. Select the Product entity and view the Mapping Details window. Click Maps to Product and select <Delete> to delete the table mapping.
10. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor. This will close the designer and open the .edmx file in the XML Editor.
11. Add the code in Listing 15-2 to the .edmx file immediately after the **<EntityContainerMapping>** tag in the mapping section.

Listing 15-2. QueryView to customize the Product entity

```
<EntitySetMapping Name="Products">
          <QueryView>
                  select value
                  EFRecipesModel.Product(p.ProductId,
                  p.Name,
                  case when p.StockCount is null then 0
                  else p.StockCount
                  end,
                  case when p.Discontinued = 1 then True
                  else False
                  end,
                  case when p.Discontinued = 0 and s.Name == "CallComm" then True
                  else False
                  end,
                  case when s.Name is null then "Unknown"
                  else s.Name
                  end
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                )
                from EFRecipesModelStoreContainer.Product as p
                left join EFRecipesModelStoreContainer.Supplier as s
                on p.SupplierId = s.SupplierId
        </QueryView>
</EntitySetMapping>
```

The resulting model should look like the one in Figure 15-4.

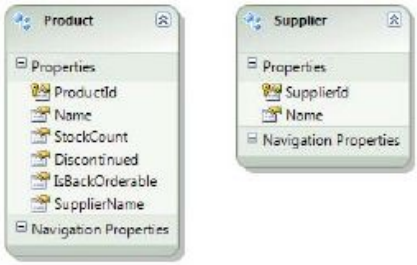


Figure 15-4. *The completed model with our QueryView-based Product entity and the Supplier entity*

How It Works

We used QueryView together with changes to the conceptual model to map and reshape the columns from the Product and Supplier tables to a more consumable Product entity. This reshaping included changing the data types for some properties, ignoring a column, and creating computed properties.

In the conceptual model, we changed the maximum name length to 50 characters. We changed the data type of the Discontinued property from Int32 to Boolean. And we changed the StockCount to not accept nulls.

In the Entity SQL expression (in the QueryView), we handle the case of existing null stock counts by changing the value to 0. We change the Discontinued values of 1 and 0 to **true** and **false**, respectively. We compute the value of the IsBackorderable property if the product is not discontinued and the supplier is 'CallCom.' Finally, we set the Supplier name, if it exists, from the Supplier table. If the name is null, then we substitute the string 'Unknown.'

The code in Listing 15-3 demonstrates inserting into and retrieving from our model. Because we used QueryView and did not supply stored procedures for the Insert, Update, and Delete actions, our Product entity is read-only. In the code of Listing 15-3, we use the **ExecuteStoreCommand()** method to directly execute the appropriate insert statements.

Listing 15-3. Inserting and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var sup1 = new Supplier { SupplierId = 1, Name = "CallCom" };
    var sup2 = new Supplier { SupplierId = 2, Name = "Toys, Ltd." };
    context.Suppliers.AddObject(sup1);
    context.Suppliers.AddObject(sup2);
    context.SaveChanges();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    // insert some products directly
    context.ExecuteStoreCommand(@"insert into chapter15.product(productid,name,
                                 description,stockcount,discontinued)
                                 values (1,'Flowers','Dozen red roses',4,1)");
    context.ExecuteStoreCommand(@"insert into chapter15.product(productid,name
                         description,stockcount,discontinued,supplierid)
                         values (2,'Red Fire Truck',null,null,0,1)");
}

using (var context = new EFRecipesEntities())
{
    foreach (var p in context.Products)
    {
        Console.WriteLine("\nName: {0}", p.Name);
        Console.WriteLine("Stock Count: {0}", p.StockCount.ToString());
        Console.WriteLine("Discountinued: {0}", p.Discontinued ? "Yes" : "No");
        Console.WriteLine("Supplier: {0}", p.SupplierName);
    }
}
```

The output from the code in Listing 15-3 is the following:

```
Name: Flowers

Stock Count: 4

Discountinued: Yes

Supplier: Unknown



Name: Red Fire Truck

Stock Count: 0

Discountinued: No

Supplier: CallCom
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

15-3. Creating Conditional Associations

Problem

You have a link table that joins two tables in a many-to-many relationship. The link table contains a column that defines the role for the relationship. You want each role in the relationship to surface as a unique association between the two entities.

Solution

Suppose you have a couple of tables in a many-to-many relationship with a link table, as shown in the database diagram in Figure 15-5.

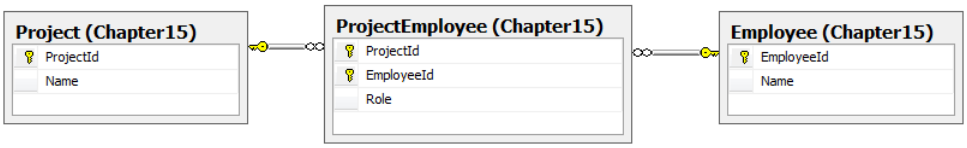


Figure 15-5. *Project and Employee in a many-to-many relationship with a column designating the role for the relationship*

A project can be associated with many employees, and an employee can be part of many projects. Additionally, for each project and employee relationship, the link table contains a column that indicates the role the employee plays in the related project. In our example, an employee can be a simple project member with no special responsibilities or a project manager, with all the responsibilities of the role.

In the model, we want to expose each role as a separate association. This means we would have an association for members and an association for project managers. To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Project, ProjectEmployee, and Employee tables. Or update an existing model with these tables.
2. Create the stored procedures in Listing 15-4 in the database. Once the database has these new stored procedures, right-click the design surface and select Update Model from Database. Select the stored procedures and click Add.
3. Right-click the ProjectEmployee entity and select Delete. Select No when prompted to delete the entity from the store model. This will delete the ProjectEmployee entity from the conceptual model, but leave the now unmapped entity in the store model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

4. Right-click the design surface and select Add ➤ Association. Name the new association **Members**. Set the multiplicity on both ends to be many. This will create a many-to-many association. Name the navigation properties **Members** on the Project side and **MemberProjects** on the Employee side.
5. Right-click the design surface and select Add ➤ Association. Name the new association ProjectManagers. Set the multiplicity on the Project side to many and the multiplicity on the Employee side to one. Name the navigation properties **ManagerProjects** on the Employee side and **ProjectManager** on the Project side.
6. Now we need to map our stored procedures to the Insert, Update, and Delete actions. Select the Employee entity and view the Mapping Details window. Click the Map Entity to Functions button on the bottom left of the Mapping Details window. Map the stored procedures to actions, as shown in Figure 15-6. Be sure to map the result column EmployeeId to the EmployeeId property. Entity Framework will use the stored generated EmployeeId for the entity key.
7. Map the stored procedures to the Insert, Update, and Delete actions for the Project entity. As with the Employee entity, be sure to map the result column ProjectId to the ProjectId property.
8. Select the Employee entity and view the Mapping Details window. Click Maps to Employee and select <Delete> to delete the mapping to the Employee table.
9. Select the Project entity and view the Mapping Details window. Click Maps to Project and select <Delete> to delete the mapping to the Project table.

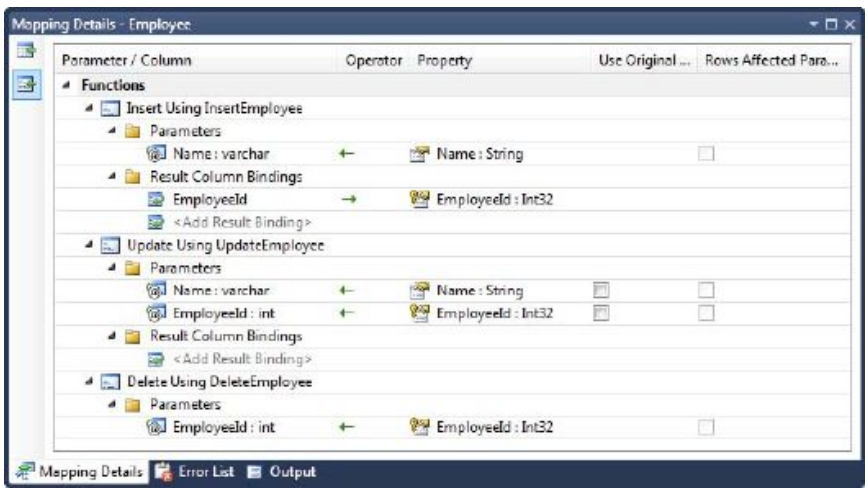


Figure 15-6. *Mapping the stored procedures to the Insert, Update, and Delete actions for the Employee entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

At this point, we have configured the conceptual model. The next steps create the conditional association. Modeling conditional associations is not directly supported by the designer. We will map the association using QueryView. As you have seen in other recipes, when we use QueryView, all the related entities must also be mapped using QueryView. Caution: After making these changes to the .edmx file, you will no longer be able to open the file in the designer.

10. Insert the QueryView in Listing 15-5 right after the **`<EntitySetMapping Name="Projects">`** tag in the C-S mapping layer.
11. Insert the QueryView in Listing 15-6 right after the **`<EntitySetMapping Name="Employees">`** tag in the C-S mapping layer.
12. Insert the code in Listing 15-7 after the **`<EntityContainerMapping>`** tag. These two QueryViews map a row in the link table to either the Member association or the ProjectManager association based on the Role column.

Listing 15-4. Stored procedures for the Insert, Update, and Delete actions for the entities (notice that the insert procedures for the Project and Employee tables returns the newly generated Id value)

```
create procedure [chapter15].[InsertProject]
(@Name varchar(50))
as
begin
        insert into Chapter15.Project(Name) values (@Name)
        select SCOPE_IDENTITY() as ProjectId
end
go

create procedure [chapter15].[UpdateProject]
(@Name varchar(50), @ProjectId int)
as
begin
        update Chapter15.Project set Name = @Name where ProjectId = @ProjectId
end
go

create procedure [chapter15].[DeleteProject]
(@ProjectId int)
as
begin
        delete Chapter15.Project where ProjectId = @ProjectId
end
go

create procedure [chapter15].[InsertEmployee]
(@Name varchar(50))
as
begin
        insert Chapter15.Employee (Name) values (@Name)
        select SCOPE_IDENTITY() as EmployeeId
end
go
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
create procedure [chapter15].[UpdateEmployee]
(@Name varchar(50), @EmployeeId int)
as
begin
        update Chapter15.Employee set Name = @Name where EmployeeId = @EmployeeId
end
go

create procedure [chapter15].[DeleteEmployee]
(@EmployeeId int)
as
begin
        delete Chapter15.Employee where EmployeeId = @EmployeeId
end
go

create procedure [chapter15].[InsertMember]
(@ProjectId int, @EmployeeId int)
as
begin
        insert into Chapter15.ProjectEmployee values (@ProjectId, @EmployeeId, 'MM')
end
go

create procedure [chapter15].[DeleteMember]
(@ProjectId int, @EmployeeId int)
as
begin
        delete Chapter15.ProjectEmployee where ProjectId = @ProjectId and EmployeeId =
@EmployeeId
end
go

create procedure [chapter15].[InsertProjectManager]
(@ProjectId int, @EmployeeId int)
as
begin
        insert into Chapter15.ProjectEmployee values (@ProjectId, @EmployeeId, 'PM')
end
go

create procedure [chapter15].[DeleteProjectManager]
(@ProjectId int, @EmployeeId int)
as
begin
        delete Chapter15.ProjectEmployee where ProjectId = @ProjectId and EmployeeId =
@EmployeeId
end
go
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 15-5. QueryView and procedure mapping for the Project entity

```
<QueryView>
  select value EFRecipesModel.Project(p.ProjectId,p.Name)
  from EFRecipesModelStoreContainer.Project as p
</QueryView>
```

Listing 15-6. QueryView and procedure mapping for the Employee entity

```
<QueryView>
  select value EFRecipesModel.Employee(e.EmployeeId,e.Name)
  from EFRecipesModelStoreContainer.Employee as e
</QueryView>
```

Listing 15-7. QueryView and procedure mappings for the associations

```
<AssociationSetMapping TypeName="EFRecipesModel.Members" Name="Members">
   <QueryView>
     select value EFRecipesModel.Members(
     createref(EFRecipesEntities.Employees,row(pe.EmployeeId)),
     createref(EFRecipesEntities.Projects,row(pe.ProjectId))
     )
     from EFRecipesModelStoreContainer.ProjectEmployee as pe
     where pe.Role = 'MM'
   </QueryView>
   <ModificationFunctionMapping>
     <InsertFunction FunctionName="EFRecipesModel.Store.InsertMember">
       <EndProperty Name="Project">
         <ScalarProperty Name="ProjectId" ParameterName="ProjectId" />
       </EndProperty>
       <EndProperty Name="Employee">
         <ScalarProperty Name="EmployeeId" ParameterName="EmployeeId" />
       </EndProperty>
     </InsertFunction>
     <DeleteFunction FunctionName="EFRecipesModel.Store.DeleteMember">
       <EndProperty Name="Project">
         <ScalarProperty Name="ProjectId" ParameterName="ProjectId" />
       </EndProperty>
       <EndProperty Name="Employee">
         <ScalarProperty Name="EmployeeId" ParameterName="EmployeeId" />
       </EndProperty>
     </DeleteFunction>
   </ModificationFunctionMapping>
 </AssociationSetMapping>
 <AssociationSetMapping TypeName="EFRecipesModel.ProjectManager" Name="ProjectManager">
   <QueryView>
     select value EFRecipesModel.ProjectManager(
     createref(EFRecipesEntities.Employees,row(pe.EmployeeId)),
     createref(EFRecipesEntities.Projects,row(pe.ProjectId))
     )
     from EFRecipesModelStoreContainer.ProjectEmployee as pe
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
      where pe.Role = 'PM'
    </QueryView>
    <ModificationFunctionMapping>
      <InsertFunction FunctionName="EFRecipesModel.Store.InsertProjectManager">
        <EndProperty Name="Project">
          <ScalarProperty Name="ProjectId" ParameterName="ProjectId" />
        </EndProperty>
        <EndProperty Name="Employee">
          <ScalarProperty Name="EmployeeId" ParameterName="EmployeeId" />
        </EndProperty>
      </InsertFunction>
      <DeleteFunction FunctionName="EFRecipesModel.Store.DeleteMember">
        <EndProperty Name="Project">
          <ScalarProperty Name="ProjectId" ParameterName="ProjectId" />
        </EndProperty>
        <EndProperty Name="Employee">
          <ScalarProperty Name="EmployeeId" ParameterName="EmployeeId" />
        </EndProperty>
      </DeleteFunction>
    </ModificationFunctionMapping>
  </AssociationSetMapping>
```

How It Works

That's a lot of code. We used QueryView for all the mappings, which made for a lot of small stored procedures and some tedious XML changes to the .edmx file. For the Members association, we used QueryView (see Figure 15-8) to map the ProjectEmployee table. The constructor takes two parameters. The first parameter is a reference to the Project entity. The second parameter is a reference to the Employee entity. To get the references to the entities, we use the Entity SQL operator **`createref()`**. This operator takes the fully qualified name of the EntitySet and an entity key. Notice that we also applied a filter on the ProjectEmployee table where **`Role == 'MM'`**. This filter limits the results to employees in the Member relationship to the project.

The ProjectManager relationship is implemented in the same way except for the filter, **`Role == 'PM'`**, which limits the results to the employee in the ProjectManager relationship to the project.

Mapping an entity based on conditions is natively supported by Entity Framework. This is the basis for modeling Table per Hierarchy inheritance. Mapping an association based on conditions is not supported and requires QueryView as we demonstrated in this example. The code in Listing 15-8 illustrates how to insert and query our model.

Listing 15-8. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var proj = new Project { Name = "Highway 101 Access Route" };
    proj.Members.Add(new Employee { Name = "Jim Stone" });
    proj.Members.Add(new Employee { Name = "Roland Jones" });
    proj.Members.Add(new Employee { Name = "Jennifer Collins" });
    proj.ProjectManager = new Employee { Name = "Sue Raven" };
    context.Projects.AddObject(proj);
    context.SaveChanges();
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    foreach (var p in context.Projects)
    {
        Console.WriteLine("Project: {0}, Manager: {1}", p.Name, p.ProjectManager.Name);
        Console.WriteLine("Members:");
        foreach (var m in p.Members)
        {
            Console.WriteLine("\t{0}", m.Name);
        }
    }
}
```

The following is the output from the code in Listing 15-8:

```
Project: Highway 101 Access Route, Manager: Sue Raven

Members:

        Roland Jones

        Jim Stone

        Jennifer Collins
```

15-4. Fabricating Additional Inheritance Hierarchies

Problem

You have a table you want to model using Table per Hierarchy inheritance and you want to add one or more additional hierarchies that are not represented in the database.

Solution

Let's say your table looks like the one in Figure 15-7.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

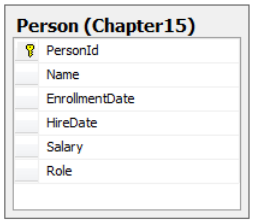


Figure 15-7. *The Person table with information about an individual in a school*

The table in Figure 15-7 holds information about students, instructors, and principals in a school. We want to use Table per Hierarchy inheritance to model this table as a hierarchy of staff and students. Students, instructors, or principals are distinguished in the table by the role column. We will use the role column as our discriminator in the model. We want to introduce a Staff type in our hierarchy so that we can group and ultimately program against, staff and students, separately. Representing the Staff level in the hierarchy is the key part of the problem.

To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Person table. Or update an existing model with this table.
2. Remove the Role property from the Person entity.
3. Right-click the Person entity and select properties. Set the Abstract property to True. This marks the Person entity as abstract.
4. Right-click the design surface and select Add ➤ Entity. Name the new entity **Student** and set its base type to Person.
5. Move the EnrollmentDate property from the Person entity to the Student entity. You can use Cut/Paste to move the property.
6. Select the Student entity and view the Mapping Details window. Select Add a Table or View and choose the Person table.
7. In the Mapping Details window, add a condition for the Student entity When `Role = Student`. This maps the Student entity when the Role is Student.
8. Right-click the design surface and select Add ➤ Entity. Name the new entity **Staff** and set its base type to Person.
9. Move the HireDate and Salary properties from the Person entity to the Staff entity.
10. Right-click the Staff entity and select Properties. Set the Abstract property to True. This marks the Staff entity as abstract.
11. Right-click the design surface and select Add ➤ Entity. Name the new entity **Principal** and set its base type to Staff.
12. Select the Principal entity and view the Mapping Details window. Select Add a Table or View and choose the Person table.

Licensed to Matt Allen 2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

13. In the Mapping Details window, add a condition for the Principal entity When **`Role = Principal`**. This maps the Principal entity when the Role is Principal.

14. Right-click the design surface and select Add ➤ Entity. Name the new entity **Instructor** and set its base type to Staff.

15. Select the Instructor entity and view the Mapping Details window. Select Add a Table or View and choose the Person table.

16. In the Mapping Details window, add a condition for the Instructor entity When **`Role = Instructor`**. This maps the Instructor entity when the Role is Instructor.

At this point, we have the inheritance hierarchy in place with all the entities. We have marked Person and Staff as abstract because we will never create them directly. We have not yet mapped the properties of the Principal or the Instructor entities. This is not supported by the designer, so we need to make some minor changes to the .edmx file.

17. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor. This will close the designer and open the .edmx file in the XML editor.

18. Add the HireDate and Salary mappings to both the Principal and Instructor entities. The XML in Listing 15-9 highlights the required changes.

Listing 15-9. Mapping the HireDate and Salary properties in the Principal and Instructor entities

```
<EntityContainerMapping StorageEntityContainer="EFRecipesModelStoreContainer"
                        CdmEntityContainer="EFRecipesEntities">
  <EntitySetMapping Name="People">
    <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Person)">
      <MappingFragment StoreEntitySet="Person">
        <ScalarProperty Name="PersonId" ColumnName="PersonId" />
        <ScalarProperty Name="Name" ColumnName="Name"/>
      </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Student)">
      <MappingFragment StoreEntitySet="Person">
        <ScalarProperty Name="PersonId" ColumnName="PersonId" />
        <ScalarProperty Name="EnrollmentDate" ColumnName="EnrollmentDate" />
        <Condition ColumnName="Role" Value="Student" />
      </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Principal)">
      <MappingFragment StoreEntitySet="Person" >
        <ScalarProperty Name="PersonId" ColumnName="PersonId"/>
        <ScalarProperty Name="HireDate" ColumnName="HireDate"/>
        <ScalarProperty Name="Salary" ColumnName="Salary"/>
        <Condition ColumnName="Role" Value="Principal" />
      </MappingFragment>
    </EntityTypeMapping>
    <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Instructor)">
      <MappingFragment StoreEntitySet="Person" >
        <ScalarProperty Name="PersonId" ColumnName="PersonId"/>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        <ScalarProperty Name="HireDate" ColumnName="HireDate"/>
        <ScalarProperty Name="Salary" ColumnName="Salary"/>
        <Condition ColumnName="Role" Value="Instructor" />
      </MappingFragment>
    </EntityTypeMapping>
  </EntitySetMapping>
</EntityContainerMapping>
```

The resulting model is shown in Figure 15-8.

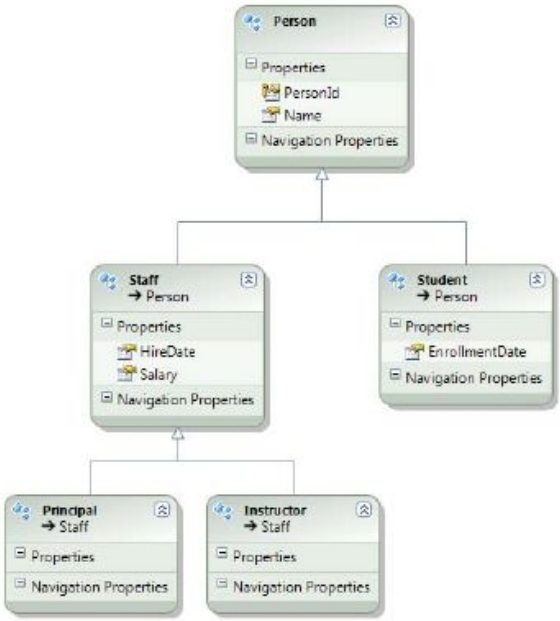


Figure 15-8. *The model with the Staff entity added to the hierarchy*

How It Works

We added the Staff entity that introduced a new level in the inheritance hierarchy. This allows us to better organize the hierarchy and simplify the programming because the Staff entity contains all the properties shared among the Principal and Instructor entities.

This modeling scenario is not directly supported by the designer. We had to make some minor changes (as shown in Listing 15-9) to the .edmx file to map the properties of the Principal and Instructor entities.

The code in Listing 15-10 demonstrates inserting into and retrieving from our model.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Listing 15-10. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var p = new Principal { Name = "Jill Robins",
                            HireDate = DateTime.Parse("8/12/2002"),
                            Salary = 72500M };
    var i1 = new Instructor { Name = "Roland Jones",
                            HireDate = DateTime.Parse("8/14/2005"),
                            Salary = 61000M};
    var i2 = new Instructor { Name = "Steven Curtis",
                            HireDate = DateTime.Parse("8/23/1992"),
                            Salary = 68200M };
    context.People.AddObject(new Student { Name = "Karen Roberts" });
    context.People.AddObject(new Student {Name = "Bobby McGivens"});
    context.People.AddObject(new Student {Name = "Janis Hettler"});
    context.People.AddObject(p);
    context.People.AddObject(i1);
    context.People.AddObject(i2);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Staff");
    Console.WriteLine("=====");
    foreach (var staff in context.People.OfType<Staff>())
    {
        Console.WriteLine("\t{0}, Hire date: {1}, Salary: {2} {3})",
                          staff.Name, staff.HireDate.Value.ToShortDateString(),
                          staff.Salary.Value.ToString("C"),
                          staff is Principal ? "Principal" : "Instructor");
    }
    Console.WriteLine("\nStudents");
    Console.WriteLine("==========");
    foreach (var student in context.People.OfType<Student>())
    {
        Console.WriteLine("\t{0}", student.Name);
    }
}
```

The output from the code in Listing 15-10 is the following:

```
Staff

=====

        Jill Robins, Hire date: 8/12/2002, Salary: $72,500.00 Principal)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        Roland Jones, Hire date: 8/14/2005, Salary: $61,000.00 Instructor)

        Steven Curtis, Hire date: 8/23/1992, Salary: $68,200.00 Instructor)


Students

==========

        Karen Roberts

        Bobby McGivens

        Janis Hettler
```

15-5. Sharing Audit Fields Across Multiple Entities

Problem

You have several tables with common audit columns and you want to create a model that simplifies updating these audit columns

Solution

Let's say you have the two tables shown in Figure 15-9. These tables share the CreateDate and ModifiedDate audit columns. We want to create a model that factors these common fields into a base entity so that we can simplify tracking the audit information.

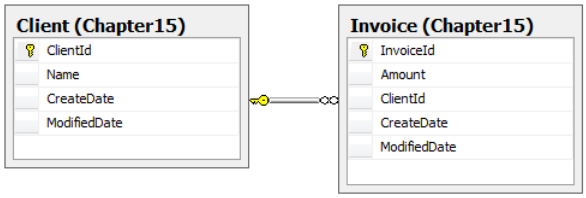


Figure 15-9. *Client and Invoice tables share the CreateDate and ModifiedDate audit columns*

To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Client and Invoice tables. Or update an existing model with these tables.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

2. Right-click the design surface and select Add ➤ Entity. Name the new entity **Audit** and check the Create key property box to create an entity key with the Id property.
3. Move the audit fields CreateDate and ModifiedDate from the Client entity to the Audit entity. You can use Cut/Paste to move these. Delete these audit fields from the Invoice entity.
4. Right-click the Audit entity and select Add ➤ Inheritance. Select the Audit entity as the base entity and the Client entity as the derived entity.
5. Repeat Step 4, selecting Audit as the base entity and Invoice as the derived entity.
6. Remove the ClientId from the Client entity. Remove the InvoiceId properties from the Invoice entity. We will use the Id property in the Audit entity as the entity key.
7. Right-click the Audit entity and select properties. Change the Abstract property to True. This marks the Audit entity as abstract.
8. Right-click the association between the Client entity and the Invoice entity. View the Properties of the association. Click the box in the Referential Constraints. Set the Principal as Client. Set the Dependent Property to the ClientId property. The dialog box should look like the one in Figure 15-10.
9. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor. Edit the entity set mapping in the Mapping layer as shown in the highlighted parts in Listing 15-11.

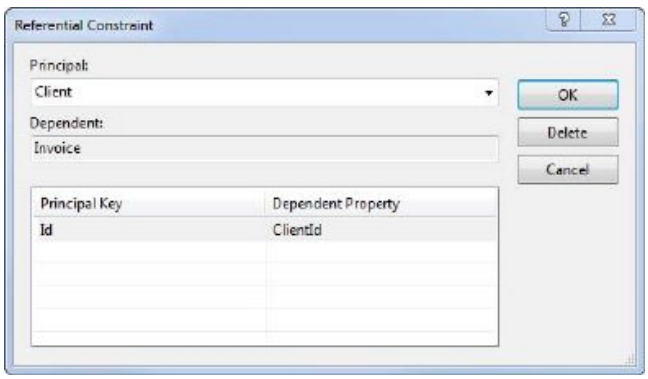


Figure 15-10. *The Referential Constraint for the Foreign Key association between the Client and Invoice entities*

Listing 15-11. Entity set mappings for the Audits

```
<EntitySetMapping Name="Audits">
  <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Client)">
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    <MappingFragment StoreEntitySet="Client">
      <ScalarProperty Name="Id" ColumnName="ClientId" />
      <ScalarProperty Name="Name" ColumnName="Name" />
      <ScalarProperty Name="CreateDate" ColumnName="CreateDate" />
      <ScalarProperty Name="ModifiedDate" ColumnName="ModifiedDate" />
    </MappingFragment>
  </EntityTypeMapping>
  <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Invoice)">
    <MappingFragment StoreEntitySet="Invoice">
      <ScalarProperty Name="Id" ColumnName="InvoiceId" />
      <ScalarProperty Name="ClientId" ColumnName="ClientId" />
      <ScalarProperty Name="Amount" ColumnName="Amount" />
      <ScalarProperty Name="CreateDate" ColumnName="CreateDate" />
      <ScalarProperty Name="ModifiedDate" ColumnName="ModifiedDate" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

If the ClientId and InvoiceId columns, which alternately serve as the entity key for the Audit entity, do not overlap, the model is complete. The conflict here is that both the Client and Invoice entities are part of the Audits entity set. The entity key must be unique across this entity set. The entity key comes from both the ClientId column and the InvoiceId column. If they overlap, it is possible we would have non-unique keys in the Audits entity set. If these keys are naturally unique because they are drawn from two non-overlapping sets, then we are safe. This would be the case if, for example, both are identity columns with Client enumerating odd values (seed of 1, increment of 2) and Invoice enumerating even values (seed of 2, increment of 2).

If ClientId and InvoiceId could conflict, then we need to remove the Audits entity set and move the Client and Invoice entities to their own entity sets. This involves converting the model so that the Audit entity is in both of the new entity sets. The use of multiple entity sets per type, better known as *Multiple Entity Sets per Type (MEST)*, is not supported in the current designer. Converting the model to use MEST will prevent the designer from opening the model. To move the Client and Invoice entities to their own entity sets and represent the Audit entity in these two sets using MEST, continue with these steps:

10. Right-click the .edmx file and select Open With ➤ XML Editor. In the conceptual layer, replace the Audits definition with the XML in Listing 15-12.
11. Removing the Audits broke the association in our model. In the conceptual layer, replace the AssociationSet definition with the code in Listing 15-13.
12. Update the mapping layer with by replacing the entity set mapping for the Audits with the code in Listing 15-14.

Listing 15-12. Replace the <EntitySet Name="Audits"…> tag with this code

```
<EntitySet Name="Clients" EntityType="EFRecipesModel.Client" />
<EntitySet Name="Invoices" EntityType="EFRecipesModel.Invoice" />
```

Listing 15-13. Replace the <AssociationSet Name="FK_Invoice_Client …> code with this code

```
<AssociationSet Name="FK_Invoice_Client"
                Association="EFRecipesModel.FK_Invoice_Client">
  <End Role="Client" EntitySet="Clients" />
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
  <End Role="Invoice" EntitySet="Invoices" />
</AssociationSet>
```

Listing 15-14. Replace mappings for the Client and Invoice entities with this code

```
<EntitySetMapping Name="Clients">
  <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Client)">
    <MappingFragment StoreEntitySet="Client">
      <ScalarProperty Name="Name" ColumnName="Name" />
      <ScalarProperty Name="Id" ColumnName="ClientID" />
      <ScalarProperty Name="CreateDate" ColumnName="CreateDate" />
      <ScalarProperty Name="ModifiedDate" ColumnName="ModifiedDate" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
<EntitySetMapping Name="Invoices">
  <EntityTypeMapping TypeName="IsTypeOf(EFRecipesModel.Invoice)">
    <MappingFragment StoreEntitySet="Invoice">
      <ScalarProperty Name="Amount" ColumnName="Amount" />
      <ScalarProperty Name="Id" ColumnName="InvoiceID" />
      <ScalarProperty Name="CreateDate" ColumnName="CreateDate" />
      <ScalarProperty Name="ModifiedDate" ColumnName="ModifiedDate" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

The model is shown in Figure 15-11.

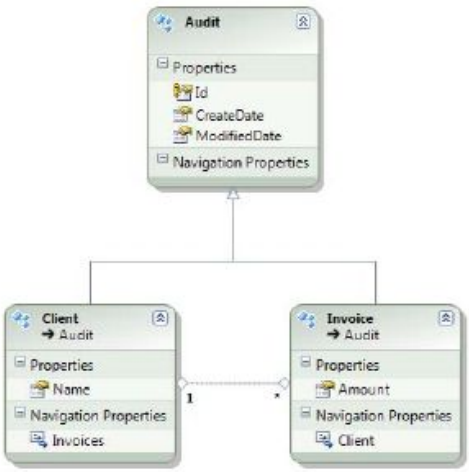


Figure 15-11. The completed model with the Audit entity holding the common audit fields

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Our model uses Table per Concrete Type inheritance to factor the audit fields into the Audit base entity. In the typical scenario for Table per Concrete Type inheritance, the tables for the derived types share most of the same schema. In this example, only the audit fields are shared.

As is common with this inheritance model, we need to address the problem of ensuring that the keys from the tables for the derived types do not overlap. Because the base entity's key comes from these tables, any overlap violates the uniqueness requirement for entity keys in an entity set. One way to address this problem is to move the derived entities into their own entity sets. This solves the key uniqueness problem (assuming, of course, that the keys are unique within each table). Unfortunately, moving the entities into separate entity sets means that the Audit entity (the base entity) now spans two entity sets. MEST is not supported by the current designer. After applying these changes, the model will no longer open in the designer.

Regardless of how you choose to solve the entity key problem, once the base entity has the audit fields, we can simplify maintaining the audit information for the derived entities. The code in Listing 15-15 (this is the non-MEST version) demonstrates inserting into and retrieving from our model. The code implements a handler for the **SavingChanges** event. In this handler, we find all the objects that are newly created or modified. For each of these, we set the appropriate audit fields. This approach provides a clean separation between the main code that inserts and updates the model and the code that manages the audit fields.

Chapter 12 covers a number of recipes involving the **SavingChanges** event.

Listing 15-15. Inserting into and retrieving from our model (this is the TPC without MEST version; the MEST version would use the Clients and Invoices sets rather than the AuditSet set for adding the objects)

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
        {
            var c1 = new Client { Name = "Joanne Wise" };
            var c2 = new Client { Name = "Robert Marr" };
            var c3 = new Client { Name = "Shelly King" };
            var i1 = new Invoice { Amount = 99.23M };
            var i2 = new Invoice { Amount = 29.95M };
            c1.Invoices.Add(i1);
            c3.Invoices.Add(i2);
            context.Audits.AddObject(c1);
            context.Audits.AddObject(c2);

            context.Audits.AddObject(c3);
            context.SaveChanges();
            Console.WriteLine("Waiting 10 seconds to update...");
            System.Threading.Thread.Sleep(10 * 1000);
            i1.Amount = 98.49M;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
            i2.Amount = 39.99M;
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            context.ContextOptions.LazyLoadingEnabled = true;
            Console.WriteLine("Invoices...");
            foreach (var bill in context.Audits.OfType<Invoice>())
            {
                Console.WriteLine("{0} Amount: {1}", bill.Client.Name,
                                  bill.Amount.ToString("C"));
                Console.WriteLine("\tCreated: {0}",
                                  bill.CreateDate.ToLongTimeString());
                Console.WriteLine("\tLast Modified: {0}\n",
                                  bill.ModifiedDate.ToLongTimeString());
            }
        }
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges += (o, s) =>
        {
            var inaudits = this.ObjectStateManager
                         .GetObjectStateEntries(System.Data.EntityState.Added)
                         .Where(entry => entry.Entity is Audit)
                         .Select(entry => entry.Entity as Audit);
            foreach (var audit in inaudits)
            {
                audit.CreateDate = DateTime.Now;
                audit.ModifiedDate = DateTime.Now;
            }
            var modaudits = this.ObjectStateManager
                       .GetObjectStateEntries(System.Data.EntityState.Modified)
                       .Where(entry => entry.Entity is Audit)
                       .Select(entry => entry.Entity as Audit);
            foreach (var audit in modaudits)
            {
                audit.ModifiedDate = DateTime.Now;
            }
        };
    }
}
```

The output from the code in Listings 15-15 is the following:

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
Waiting 10 seconds to update...

Invoices...

Joanne Wise Amount: $98.49

        Created: 11:46:38 AM

        Last Modified: 11:46:49 AM


Shelly King Amount: $39.99

        Created: 11:46:38 AM

        Last Modified: 11:46:49 AM
```

Licensed by Matt Allen 2175280

15-6. Modeling a Many-to-Many Relationship with Payload

Problem

You have two tables in a many-to-many relationship. The relationship contains additional information or payload. You want to create a model that models this relationship as a many-to-many association as well as two one-to-many associations.

Solution

Let's say we have three tables like the ones shown in the database diagram in Figure 15-12.

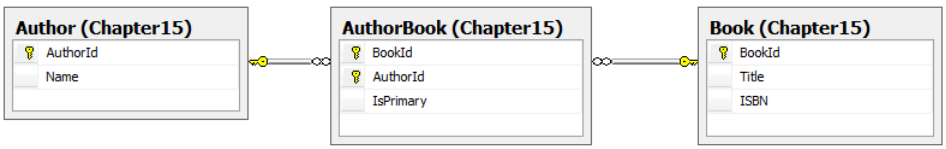


Figure 15-12. *Author and Book in a many-to-many relationship through the AuthorBook link table*

In Figure 15-12, the table AuthorBook is a link table that provides the many-to-many relationship between the Author table and the Book table. The IsPrimary column stores information about the relationship between an author and a book. This extra information is often referred to as payload.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

If the AuthorBook table did not have the IsPrimary payload column, Entity Framework would model this relationship as a many-to-many association between the Author entity and the Book entity. The AuthorBook table would be part of the association and would not surface as a separate entity. With the IsPrimary payload column, Entity Framework represents all three tables as entities with one-to-many associations.

We want to model the tables in Figure 15-12 with a many-to-many association as well as two one-to-many associations. To create this model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Author, AuthorBook, and Book tables. Or update an existing model with these tables.
2. Right-click the design surface and select Add ➤ Association. Create the association between the Author entity and the Book entity. Set the multiplicity on both ends to many. Pluralize the navigation properties by setting them to Books on the Author end and Authors on the Book end.
3. Use the code in Listing 15-16 to create a view in the database. This provides a payload-free view of the link table.
4. Right-click the design surface and select Update Model from Database. Expand Views in the Add tab and select the vwAuthorBook view. This will update the model with the vwAuthorBook view.
5. Right-click the new vwAuthorBook entity and select Delete. When prompted to Delete Unmapped Tables and Views, select no. This will remove the entity from the conceptual layer, but leave the view in the storage layer.
6. Select the many-to-many association between the Author and Book entities and view the Mapping Details window. In the Add a Table or View drop-down, select the vwAuthorBook view. You may need to scroll the control down to see the newly added view. The column and property mappings should match those in Figure 15-13.

The completed model is shown in Figure 15-14.

Listing 15-16. A view of the AuthorBook link table excluding the payload column

```
create view [Chapter15].[vwAuthorBook]
as select BookId, AuthorId from Chapter15.AuthorBook
```

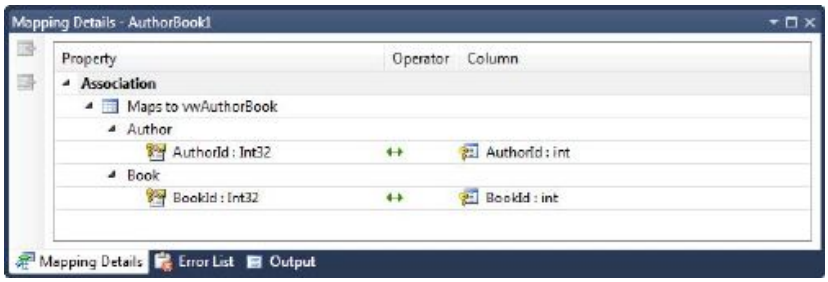


Figure 15-13. *The mappings for the many-to-many association using the vwAuthorBook view*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

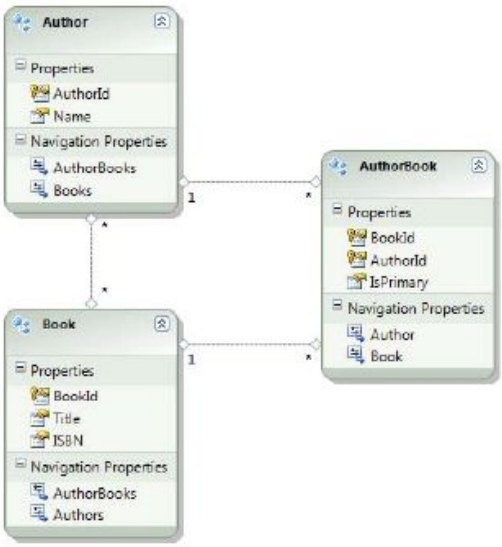


Figure 15-14. *Completed model with a read-only many-to-many association and two one-to-many associations*

How It Works

When a link table with no payload is imported into a model, it becomes part of the representation of the many-to-many association. If the link table contains payload, it surfaces as an entity and participates in two one-to-many associations. In this example, we leave the two one-to-many associations in place and introduce a many-to-many relationship using a view of the link table without the payload column. We cannot update through the many-to-many association because the association mapping relies on a (read-only) view, but we can use the many-to-many association to simplify our queries.

In the code in Listing 15-17, we use AuthorBook entity to insert into our model; then we use the many-to-many association to query the model.

Listing 15-17. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var b1 = new Book { ISBN = "978-1-847193-81-0",
                        Title = "jQuery Reference Guide" };
    var b2 = new Book { ISBN = "978-1-29298333",
                        Title = "jQuery Tips and Tricks" };
    var b3 = new Book { ISBN = "978-1033988429",
                        Title = "Silverlight 2" };
    var a1 = new Author { Name = "Jonathan Chaffer" };
    var a2 = new Author { Name = "Chad Campbell" };
    var ab1 = new AuthorBook { Author = a1, Book = b1, IsPrimary = true };
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    var ab2 = new AuthorBook { Author = a1, Book = b2, IsPrimary = false };
    var ab3 = new AuthorBook { Author = a2, Book = b3, IsPrimary = false };
    context.AuthorBooks.AddObject(ab1);
    context.AuthorBooks.AddObject(ab2);
    context.AuthorBooks.AddObject(ab3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    Console.WriteLine("Authors and Their Books...");
    foreach (var author in context.Authors)
    {
        Console.WriteLine("{0}", author.Name);
        foreach (var book in author.Books)
        {
            Console.WriteLine("\t{0}, ISBN = {1}", book.Title, book.ISBN);
        }
    }
}
```

The output from the code in Listing 15-17 is the following:

```
Authors and Their Books...

Jonathan Chaffer

        jQuery Reference Guide, ISBN = 978-1-847193-81-0

        jQuery Tips and Tricks, ISBN = 978-1-29298333

Chad Campbell

        Silverlight 2, ISBN = 978-1033988429
```

15-7. Mapping a Foreign Key Column to Multiple Associations

Problem

You have a column in a table that is a foreign key to two or more other tables. You also have a column in the table that indicates which table the foreign key references. Although most database environments do not support multiple table foreign key constraints, you want to create a model that models this structure.

We present an alternate solution to this problem in Recipe 15-8.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Solution

Suppose you have three tables like the ones shown in Figure 15-15.

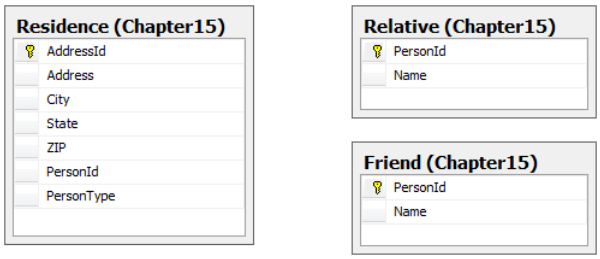


Figure 15-15. *Three related tables without foreign key constraints*

The Residence table contains the address of either a Relative or a Friend. The foreign key for either of these tables is the PersonId column in the Residence table. The PersonType column indicates whether the foreign key references the Relative table or the Friend table. There are no foreign key constraints shown in the diagram in Figure 15-15 because most database systems do not support foreign keys referencing multiple tables.

To create a model for the tables and implied relationships in Figure 15-15, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Relative and Friend tables. Or update an existing model with these tables.
2. In your database, create the vwResidence view using the code in Listing 15-18.
3. Right-click the design surface and select Update Model from Database. Add the view to the model by selecting the vwResidence view from the Views presented. Click Finish.
4. The importing process marked many of the columns of the vwResidence entity as entity keys. Except for the AddressId column, right-click each property that is marked as an entity key and select Properties. Set the Entity Key property to False.
5. Right-click the vwResidence entity and select Properties. Change the Entity Set Name to **Residences**. Change the Name to **Residence**.
6. Right-click the design surface and select Add ➤ Association. Select Residence on one end of the association with a multiplicity of many, and choose Relative on the other end with a multiplicity of zero-or-one. Change the navigation properties to Residences and Relative.
7. Right-click the association between the Relative and Residence entities and view the properties. Click the Referential Constraint box. In the dialog box, select Relative as the Principal and RelativeId as the Dependent Property. See Figure 15-16.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

8. Right-click the design surface and select Add ➤ Association. Select Residence on one end of the association with a multiplicity of many and Friend on the other end with a multiplicity of zero-or-one. Change the navigation properties to Friend and Residences.
9. Right-click the association between the Friend and Residence entities, and view the properties. Click the Referential Constraint box. In the dialog box, select Friend as the Principal and FriendId as the Dependent Property.
10. In your database, create the stored procedures for the Insert, Update, and Delete actions for the Residence entity. Use the code in Listing 15-19. Right-click the design surface and select Update Model from Database. Update your model with these new stored procedures.
11. Select each entity and view the Mapping Details window. For each, click the Map Entity to Functions button, which is on the bottom left of the Mapping Details window. Map the Insert, Update, and Delete actions to the respective stored procedures. Make sure that the Result Column Bindings map the returned value to the PersonId property for the Friend and Relative entities and the AddressId property for the Residence entity. See Figure 15-16.
12. Right-click the AddressId property in the Residence entity and view its Properties. Change the StoreGeneratedPattern property to Identity.
13. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor.
14. Removing the entity keys from the Residence entity in Step 4 only removed the keys from the conceptual level. Remove all the **<PropertyRef>** tags except for the AddressId, from Residence entity in the storage layer under the **<Key>** tag for the vwResidence EntityType.

The resulting model is shown in Figure 15-17.

Listing 15-18. A view that brings together the addresses of the relatives and friends

```
create view chapter15.vwResidence
as
select r.AddressId,r.[Address],r.City,r.[State],r.Zip,r.PersonId FriendId,null RelativeId
from chapter15.Residence as r
where r.PersonType = 'Friend'
union
select r.AddressId,r.[Address],r.City,r.[State],r.Zip,null FriendId,r.PersonId RelativeId
from chapter15.Residence as r
where r.PersonType = 'Relative'
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

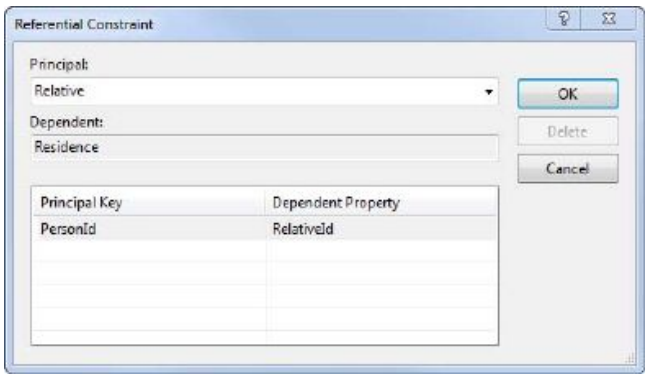


Figure 15-16. *Creating the Referential Constraint for the association between the Relative and Residence entities*

Listing 15-19. Insert, Update, and Delete actions for the Friend, Relative, and Residence entities

```
create procedure [chapter15].[InsertFriend]
(@Name varchar(50))
as
begin
        insert into Chapter15.Friend (Name) values (@Name)
        select SCOPE_IDENTITY() as PersonId
end
go

create procedure [chapter15].[UpdateFriend]
(@Name varchar(50), @PersonId int)
as
begin
        update Chapter15.Friend set Name = @Name where PersonId = @PersonId
end
go

create procedure [chapter15].[DeleteFriend]
(@PersonId int)
as
begin
        delete Chapter15.Friend where PersonId = @PersonId
end
go

create procedure [chapter15].[InsertRelative]
(@Name varchar(50))
as
begin
        insert into Chapter15.[Relative](Name) values (@Name)
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        select SCOPE_IDENTITY() as PersonId
end
go

create procedure [chapter15].[UpdateRelative]
(@Name varchar(50), @PersonId int)
as
begin
        Update Chapter15.[Relative] set Name = @Name where PersonId = @PersonId
end
go

create procedure [chapter15].[DeleteRelative]
(@PersonId int)
as
begin
        delete Chapter15.[Relative] where PersonId = @PersonId
end
go

create procedure [chapter15].[InsertResidence]
(@Address varchar(50), @City(50) varchar(50), @State varchar(20),
@Zip varchar(5), @FriendId int, @RelativeId int)
as
begin
        declare @personid int, @persontype varchar(50)
        if @FriendId is not null
        begin
                set @personid = @FriendId
                set @persontype = 'Friend'
        end
        else
        begin
                set @personid = @RelativeId
                set @persontype = 'Relative'
        end
        insert into Chapter15.Residence ([Address],City, State, ZIP,
                                         PersonId, PersonType)
        values (@Address,@City,@State,@Zip,@personid,@persontype)
        select SCOPE_IDENTITY() as AddressId
end
go

create procedure [chapter15].[UpdateResidence]
(@AddressId int, @Address varchar(50), @City varchar(50),
@State varchar(2), @Zip varchar(5), @FriendId int, @RelativeId int)
as
begin
        update Chapter15.Residence set [Address] = @Address,
        City = @City, [State] = @State, ZIP = @Zip
        where AddressId = @AddressId
end
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
go

create procedure [chapter15].[DeleteResidence]
(@AddressId int, @FriendId int, @RelativeId int)
as
begin
        delete from Chapter15.Residence where AddressId = @AddressId
end
go
```

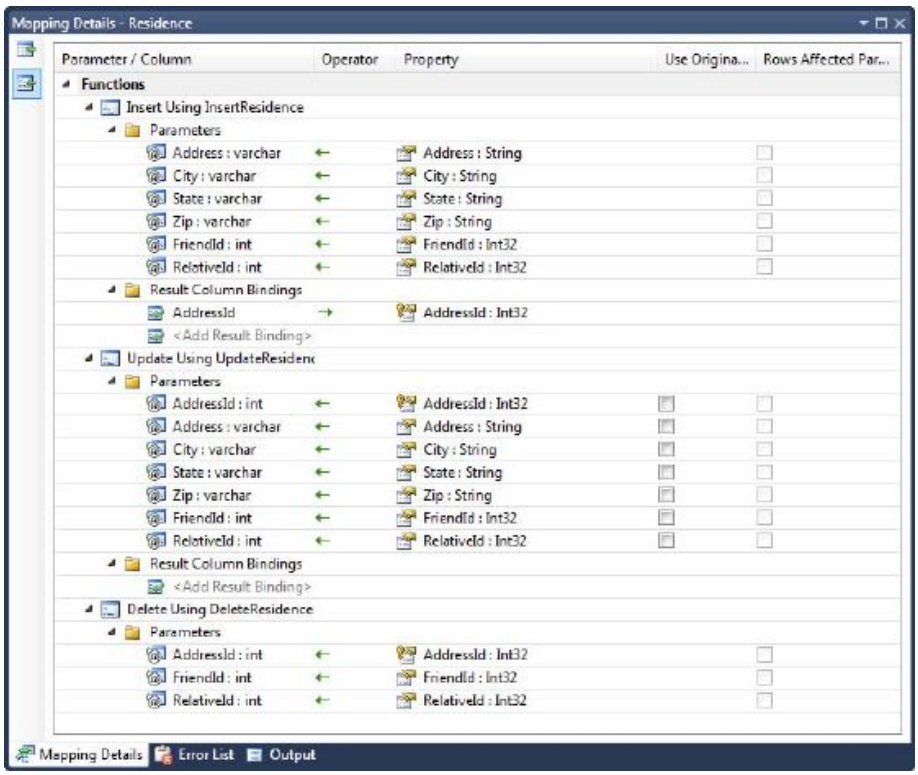


Figure 15-17. *Mapping the Insert, Update, and Delete actions to the stored procedures. Make sure the Result Column binding for the Insert action is mapped to the AddressId property.*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

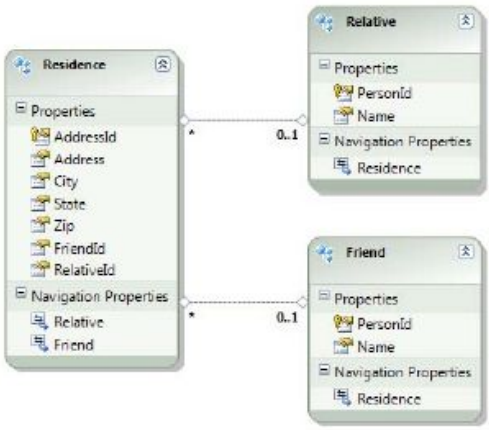


Figure 15-18. *The completed model*

How It Works

Entity Framework does not allow mapping a foreign key column to multiple associations. To model our existing database structure, we introduced a view that exposes separate foreign key columns from which we can create the model shown in Figure 15-17. These foreign key columns in our view are populated based on the values found in the PersonType column in the Residence table. If the PersonType is "Friend," the FriendId column is populated. If the PersonType is "Relative," the RelativeId column is populated.

With the introduction of the view for the Residence table, we needed to supply the implementation for the Insert, Update, and Delete actions for the entities in our model. Our implementation for these actions is shown in Listing 15-19. The mappings for the parameters and return values for these functions for the Residence entity are shown in Figure 15-17.

The code in Listing 15-20 demonstrates inserting into and retrieving from our model. One problem with this approach is that we have to be careful not to insert both a Relative and a Friend at the same residence. To guard against this sort of collision, we can handle the **SavingChanges** event and validate the changes before they are saved to the database.

Listing 15-20. Inserting into and retrieving from our model

```
class Program
{
    static void Main(string[] args)
    {
        RunExample();
    }

    static void RunExample()
    {
        using (var context = new EFRecipesEntities())
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        {
            var res1 = new Residence { Address = "123 Main", City = "Anytown",
                                       State = "CA", Zip = "90210" };
            var res2 = new Residence { Address = "1200 East Street",
                             City = "Big Town", State = "KS", Zip = "66026" };
            var f = new Friend { Name = "Joan Roland"};
            f.Residences.Add(res1);
            var r = new Relative { Name = "Billy Miner"};
            r.Residences.Add(res2);
            context.Friends.AddObject(f);
            context.Relatives.AddObject(r);
            context.SaveChanges();
        }

        using (var context = new EFRecipesEntities())
        {
            context.ContextOptions.LazyLoadingEnabled = true;
            foreach (var r in context.Residences)
            {
                if (r.Friends != null)
                    Console.WriteLine("My friend {0} lives at: ",
                                      r.Friends.Name);
                else if (r.Relatives != null)
                    Console.WriteLine("My relative {0} lives at: ",
                                      r.Relatives.Name);
                Console.WriteLine("\t{0}", r.Address);
                Console.WriteLine("\t{0}, {1} {2}", r.City, r.State,
                                  r.Zip);
            }
        }
    }
}

public partial class EFRecipesEntities
{
    partial void OnContextCreated()
    {
        this.SavingChanges += (o, s) =>
        {
            var residences =
            this.ObjectStateManager.GetObjectStateEntries(
                EntityState.Modified |
                 EntityState.Added)
                 .Where(entry => entry.Entity is Residence)
                 .Select(entry => entry.Entity as Residence);

            foreach (var residence in residences)
            {
                if ((residence.FriendId.HasValue ||
                     residence.Friends != null) &&
                    (residence.RelativeId.HasValue ||
                     residence.Relatives != null))
```


Licensed by Matt Allen 2112280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
                {
                    throw new ApplicationException("Relative or friend?");
                }
            }
        };
    }
}
```

The output of the code from Listing 15-20 is the following:

```
My friend Joan Roland lives at:

        123 Main

        Anytown, CA 90210

My relative Billy Miner lives at:

        1200 East Street

        Big Town, KS 66026
```

15-8. Using Inheritance to Map a Foreign Key Column to Multiple Associations

Problem

You have a column in a table that is a foreign key to two or more other tables. You also have a column in the table that indicates which table the foreign key references. Although most database environments do not support multiple table foreign key constraints, you want to create a model that models this structure.

We present an alternate solution to this problem in Recipe 15-7.

Solution

Suppose you have three tables like the ones shown in Figure 15-19.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

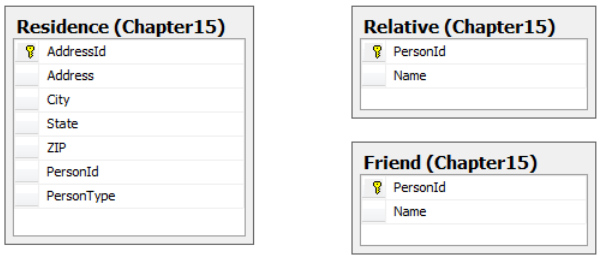


Figure 15-19. *Three related tables without foreign key constraints*

The Residence table contains the address of either a Relative or a Friend. The foreign key for either of these tables is the PersonId column in the Residence table. The PersonType column indicates if the foreign key references the Relative table or the Friend table. There are no foreign key constraints shown in the diagram in Figure 15-19 because most database systems do not support foreign keys referencing multiple tables.

In Recipe 15-7, we solved this problem using a view that exposed columns for the foreign keys for the Friend and Relative entities. In this recipe, we'll use inheritance. To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Relative, Resident, and Friend tables. Or update an existing model with these tables.
2. Delete the PersonId and PersonType properties from the Residence entity.
3. Right-click the Residence entity and view its properties. Make the entity abstract by setting the Abstract property to True.
4. Right-click the design surface and select Add ➤ Entity. Name the entity RelativeResidence. Set its base type to Residence.
5. Select the RelativeResidence entity and view the Mapping Details window. Select Residence in Add a Table or View. Add the condition `When PersonType = Relative` to the mapping.
6. Right-click the design surface and select Add ➤ Entity. Name the entity FriendResidence. Set its base type to Residence.
7. Select the FriendResidence entity and view the Mapping Details window. Select Residence in Add a Table or View. Add the condition `When PersonType = Friend` to the mapping.
8. Right-click the design surface and select Add ➤ Association. Make one end of the association the Relative entity with a multiplicity of one. Make the other end of the association the RelativeResidence entity with a multiplicity of one. Uncheck the "Add foreign key properties to 'RelativeResidence' entity" check box.
9. Select the association and view the Mapping Details window. Select Residence in Add a Table or View.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

10. Right-click the design surface and select Add ➤ Association. Make one end of the association the Friend entity with a multiplicity of one. Make the other end of the association the FriendResidence entity with a multiplicity of one. Uncheck the "Add foreign key properties to 'FriendResidence' entity" check box.
11. Select the association and view the Mapping Details window. Select Residence in Add a Table or View.
12. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor.
13. Edit the conditions for the RelativeResidence and FriendResidence association sets in the mapping sections, as shown in Listing 15-21.

Listing 15-21. AssociationSet mappings with conditions

```
<AssociationSetMapping Name="RelativeRelativeResidence"
  TypeName="EFRecipesModel.RelativeRelativeResidence"
  StoreEntitySet="Residence">
  <EndProperty Name="RelativeResidence">
    <ScalarProperty Name="AddressId" ColumnName="AddressId" />
  </EndProperty>
  <EndProperty Name="Relative">
    <ScalarProperty Name="PersonId" ColumnName="PersonId" />
  </EndProperty>
  <Condition ColumnName="PersonType" Value="Relative" />
</AssociationSetMapping>
<AssociationSetMapping Name="FriendFriendResidence"
TypeName="EFRecipesModel.FriendFriendResidence" StoreEntitySet="Residence">
  <EndProperty Name="FriendResidence">
    <ScalarProperty Name="AddressId" ColumnName="AddressId" />
  </EndProperty>
  <EndProperty Name="Friend">
    <ScalarProperty Name="PersonId" ColumnName="PersonId" />
  </EndProperty>
  <Condition ColumnName="PersonType" Value="Friend" />
</AssociationSetMapping>
```

The resulting model is shown in Figure 15-20.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

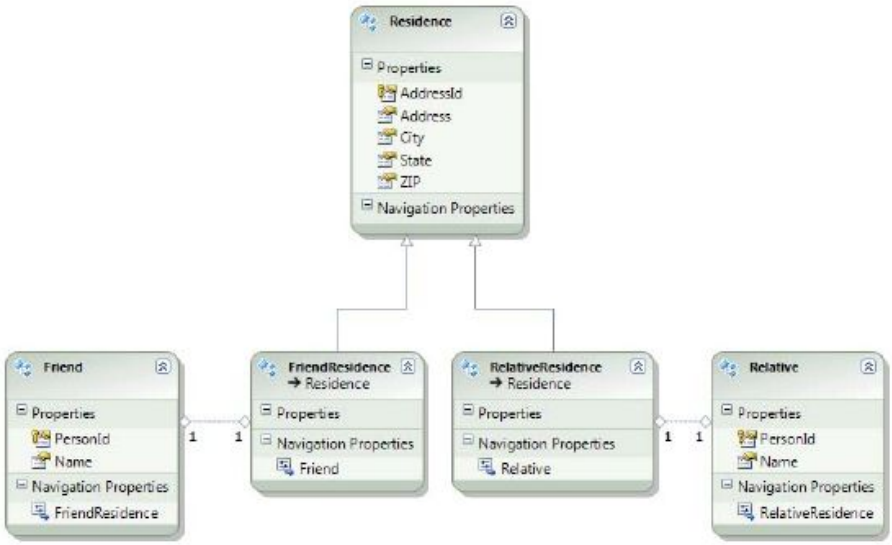


Figure 15-20. *The completed model*

How It Works

We used Table per Hierarchy inheritance in creating two derived types to represent the friend residences and the relative residences. Each of these derived types has a one-to-one conditional association to tables that contain the friends' or relatives' names.

The code in Listing 15-22 demonstrates inserting into and retrieving from our model.

Listing 15-22. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var res1 = new FriendResidence { Address = "123 Main", City = "Anytown",
                                     State = "CA", ZIP = "90210" };
    var res2 = new RelativeResidence { Address = "1200 East Street",
                            City = "Big Town", State = "KS", ZIP = "66026" };
    var f = new Friend { Name = "Joan Roland", FriendResidence = res1 };
    var r = new Relative { Name = "Billy Miner", RelativeResidence = res2 };
    context.Friends.AddObject(f);
    context.Relatives.AddObject(r);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    foreach (var r in context.Residences)
    {
        if (r is FriendResidence)
            Console.WriteLine("My friend {0} lives at: ",
                              ((FriendResidence)r).Friend.Name);
        else if (r is RelativeResidence)
            Console.WriteLine("My relative {0} lives at: ",
                              ((RelativeResidence)r).Relative.Name);
        Console.WriteLine("\t{0}", r.Address);
        Console.WriteLine("\t{0}, {1} {2}", r.City, r.State, r.ZIP);
    }
}
```

The output of the code in Listing 15-22 is the following:

```
My friend Joan Roland lives at:

        123 Main

        Anytown, CA 90210

My relative Billy Miner lives at:

        1200 East Street

        Big Town, KS 66026
```

Here we assumed that the PersonId in the Friend and Relative entities are never the same when both entities are loaded into the object context. If they are the same, an **InvalidOperationException** will be thrown.

15-9. Creating Read-only and Computed Properties

Problem

You want to create a few read-only columns on an entity as well as some computed columns.

Solution

Let's say you have an Order table with a couple of related lookup tables (see Figure 15-21). Because customers place orders, you also have a Customer table with a relationship to the Order table.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

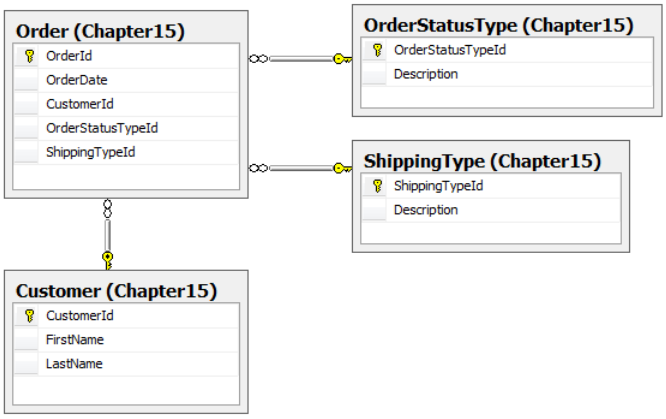


Figure 15-21. *Order with related lookup tables and Customer table*

In our model, we want the Customer entity to expose a FullName property that would be the concatenation of the FirstName column and the LastName column. Additionally, we want to expose a TotalOrders property that would be the total number of orders for the customer.

For the Orders entity, we want to include the descriptions for the order status and shipping type. These can be found in the related tables OrderStatusType and ShippingType.

To create our Entity Data Model, do the following:

1. In your database, create the views in Listing 15-23.
2. Create the stored procedures in Listing 15-24. We will use these implement the Insert, Update, and Delete actions.
3. Add a new ADO.NET Entity Data Model to your project and import the vwOrder and vwCustomer views and the stored procedures you created in Step 2. Or update an existing model with these views and stored procedures.
4. The Entity Framework Import Wizard marked many of the properties as keys in the new entities. Right-click each of these properties, except CustomerId and OrderId, and change the Entity Key Property to False.
5. Removing the entity keys in Step 4 only removed them from the conceptual layer. To remove them from the storage layer, we'll need to edit the .edmx file directly. Right-click the .edmx file and select Open With ➤ XML Editor. In the storage layer section, remove all the entity keys except for OrderId for the vwOrder entity. These are the PropertyRef tags under the Key tag. Similarly, remove all the entity keys except for CustomerId for the vwCustomer entity. Save the .edmx file and reopen it with the designer.
6. Right-click the vwCustomer entity and change the name to Customer and the entity set name to **Customers**. Right-click the vwOrder entity and change the name to **Order** and the entity set name to **Orders**.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

7. Right-click the design surface and select Add ➤ Association. Select a multiplicity of one on the Customer side and a multiplicity of many on the Order side. Rename the Order navigation property to **Orders** because it refers to a set of orders. Uncheck the "Add foreign key properties to the 'Order' entity" check box.
8. Right-click the new association and view its properties. Click in the Referential Constraint box. In the dialog box, set the Principal to Customer and the Dependent Property to CustomerId. See Figure 15-22.
9. Select the Order entity and view the Mapping Details window. In the Mapping Details window click the Map Entities to Functions button (second button on the left). Map the Order Insert, Update, and Delete stored procedures to the Insert, Update, and Delete actions. Figures 15-23 and 15-24 show the mappings for the parameters and returned values for the actions.

Listing 15-23. View that combines FirstName and LastName and computes the total orders

```
create view chapter15.vwCustomer
as
select c.*,c.FirstName + ' ' + c.LastName as FullName,
(select COUNT(*) from chapter15.[Order] where CustomerId = c.CustomerId) TotalOrders
from chapter15.Customer c
go
create view chapter15.vwOrder
as
select o.*,os.Description OrderStatus,s.Description ShippingType
from chapter15.[Order] o
join chapter15.OrderStatusType os on os.OrderStatusTypeId = o.OrderStatusTypeId
join chapter15.ShippingType s on s.ShippingTypeId = o.OrderStatusTypeId
```

Listing 15-24. Stored procedure implementations for the Insert, Update, and Delete actions for the Customer and Order entities

```
create procedure chapter15.InsertCustomer
(@FirstName varchar(50),
 @LastName varchar(50),
 @FullName varchar(50))
as
 begin
 insert into chapter15.Customer(FirstName,LastName) values (@FirstName,@LastName)
 select SCOPE_IDENTITY() CustomerId
end

go

create procedure chapter15.UpdateCustomer
 (@FirstName varchar(50),
 @LastName varchar(50),
 @FullName varchar(50),
 @CustomerId int)
as
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
begin
 update chapter15.Customer set
 FirstName = @FirstName,
 LastName = @LastName
 where CustomerId = @CustomerId
end

go

create procedure chapter15.DeleteCustomer
 (@CustomerId int)
as
begin
 delete chapter15.Customer where CustomerId = @CustomerId
end

go

create procedure chapter15.InsertOrder
 (@OrderDate date,
 @CustomerId int,
 @OrderStatusTypeId int,
 @ShippingTypeId int,
 @OrderStatus varchar(50),
 @ShippingType varchar(50))
as
begin
 insert into    chapter15.[Order](OrderDate,CustomerId,OrderStatusTypeId,ShippingTypeId)
values (@OrderDate,@CustomerId,@OrderStatusTypeId,@ShippingTypeId)
  select SCOPE_IDENTITY() OrderId
 end

go

create procedure chapter15.UpdateOrder
 (@OrderId int,
 @OrderDate date,
 @CustomerId int,
 @OrderStatusTypeId int,
 @ShippingTypeId int,
 @OrderStatus varchar(50),
 @ShippingType varchar(50))
as
begin
 update chapter15.[Order] set
 OrderDate = @OrderDate,
 CustomerId = @CustomerId,
 OrderStatusTypeId = @OrderStatusTypeId,
 ShippingTypeId = @ShippingTypeId
 where OrderId = @OrderId
end
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
go

create procedure chapter15.DeleteOrder
 (@OrderId int,
  @CustomerId int)
as
begin
 delete chapter15.[Order] where OrderId = @OrderId
 end
go
```

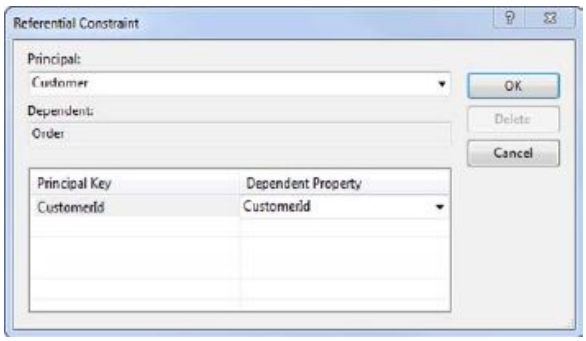


Figure 15-22. *Setting the referential constraint for the one-to-many association between the Customer and Order entities*

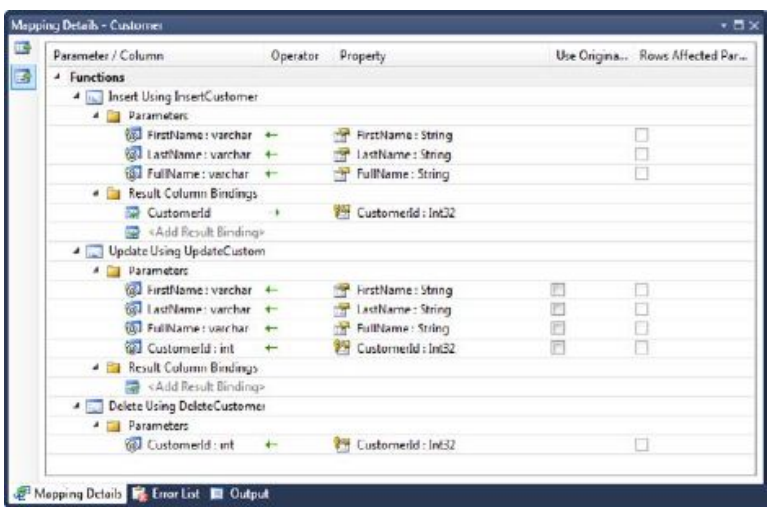
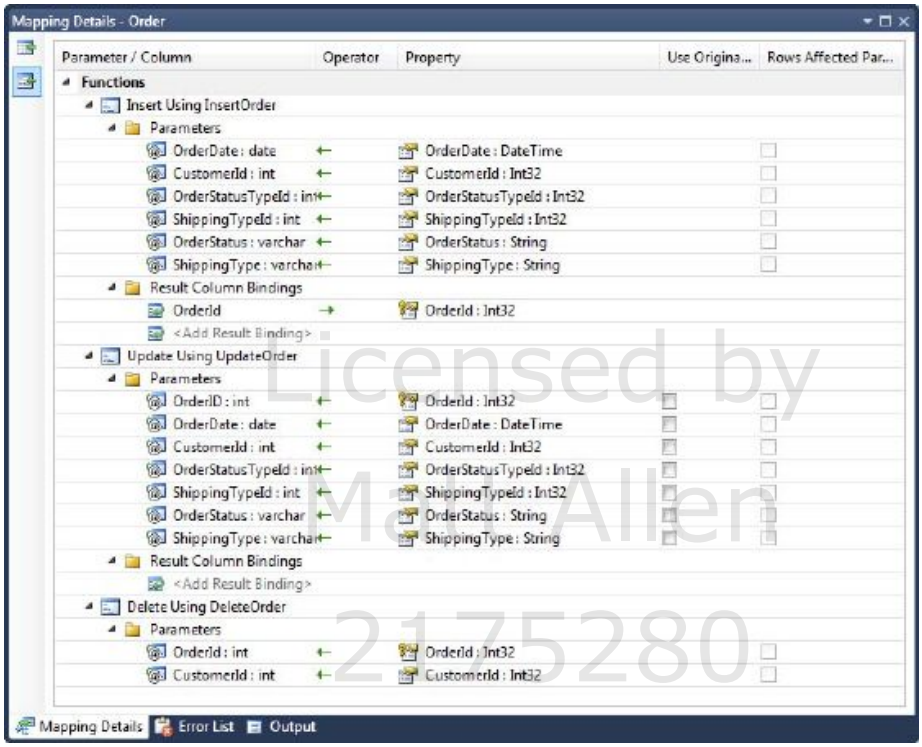


Figure 15-23. *Mappings for the parameters and returned values for the Insert, Update, and Delete actions for the Customer entity*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.



Licensed by Matt Allen 2175280

Figure 15-24. *Mappings for the parameters and returned values for the Insert, Update, and Delete actions for the Order entity*

The completed model is shown in Figure 15-25.

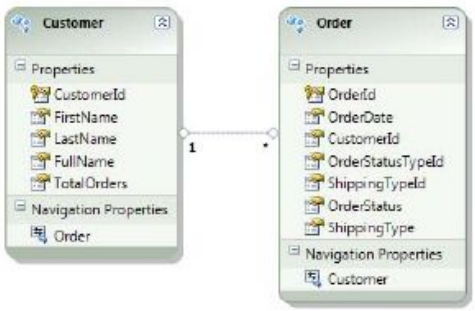


Figure 15-25. *The completed model*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

To expose the computed column TotalOrders and to combine the shipping type and order status descriptions from the lookup tables we created two views and built our model around these views. When we use views in this way, we are required to provide the Insert, Update, and Delete actions. Entity Framework has no way of automating these CRUD operations on views. The stored procedures in Listing 15-24 provide an implementation of these actions.

In the mapping of the parameters and returned values from these stored procedures, it is important to note that the returned value of the procedure mapped to the insert action must be the key for the entity. In our case, these keys were CustomerId and OrderId. Failing to map these returned values will result in a runtime error.

When we imported the views, Entity Framework marked each of the properties as part of the key for the entity. Of course, Entity Framework has no way of knowing which, if any, of the columns from the view are part of the key. As it does with tables that have no primary key constraint, it simply combines all the columns into the entity key. We removed all but the Id properties from the keys both on the design surface and in the .edmx file.

The code in Listing 15-25 demonstrates inserting into and retrieving from our model. We use code not shown in this Listing to populate the lookup tables.

Listing 15-25. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    // insert our lookup values
    context.ExecuteStoreCommand(@"insert into chapter15
               .orderstatustype(OrderStatusTypeId, Description)
               values (1,'Processing')");
    context.ExecuteStoreCommand(@"insert into chapter15
               .orderstatustype(OrderStatusTypeId, Description)
               values (2,'Shipped')");
    context.ExecuteStoreCommand(@"insert into chapter15
               .shippingtype(ShippingTypeId, Description) values (1,'UPS')");
    context.ExecuteStoreCommand(@"insert into chapter15
               .shippingtype(ShippingTypeId, Description) values (2,'FedEx')");
}

using (var context = new EFRecipesEntities())
{
    var c1 = new Customer { FirstName = "Robert", LastName = "Jones" };
    var o1 = new Order { OrderDate = DateTime.Parse("11/19/2009"),
                         OrderStatusTypeId = 2, ShippingTypeId = 1,
                         Customer = c1 };
    var o2 = new Order { OrderDate = DateTime.Parse("12/13/09"),
                         OrderStatusTypeId = 1, ShippingTypeId = 1,
                         Customer = c1 };
    var c2 = new Customer { FirstName = "Julia", LastName = "Stevens" };
    var o3 = new Order { OrderDate = DateTime.Parse("10/19/09"),
                         OrderStatusTypeId = 2, ShippingTypeId = 2,
                         Customer = c2 };
    context.Customers.AddObject(c1);
    context.Customers.AddObject(c2);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    context.ContextOptions.LazyLoadingEnabled = true;
    foreach (var c in context.Customers)
    {
        Console.WriteLine("{0} has {1} order(s)", c.FullName,
                          c.TotalOrders.ToString());
        foreach (var o in c.Orders)
        {
            Console.WriteLine("\tOrdered on: {0}",
                              o.OrderDate.ToShortDateString());
            Console.WriteLine("\tStatus: {0}", o.OrderStatus);
            Console.WriteLine("\tShip via: {0}\n", o.ShippingType);
        }
    }
}
```

The code in Listing 15-25 produces the following output:

```
Robert Jones has 2 order(s)

        Ordered on: 11/19/2009

        Status: Shipped

        Ship via: FedEx



        Ordered on: 12/13/2009

        Status: Processing

        Ship via: UPS



Julia Stevens has 1 order(s)

        Ordered on: 10/19/2009

        Status: Shipped

        Ship via: FedEx
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

15-10. Mapping an Entity to Multiple Tables

Problem

You want to insert an entity into different tables based on a property value.

Solution

Let's say you have two tables: WorkOrder and PriorityWorkOrder. The WorkOrder table contains information about common, standard priority work orders. The PriorityWorkOrder table contains the same work order information, but represents work orders that need immediate attention.

The tables might look like those in the database diagram in Figure 15-26.

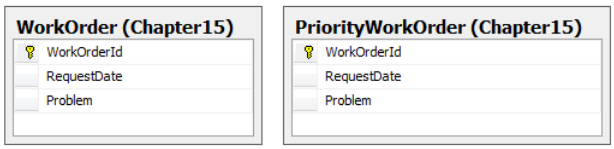


Figure 15-26. *WorkOrder and PriorityWorkOrder tables*

To create the model that maps a WorkOrder entity to these two tables, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the WorkOrder and PriorityWorkOrder tables. Or update an existing model with these tables.
2. Right-click the PriorityWorkOrder entity and select Delete. Select No in the dialog box. This will delete the entity without deleting the underlying store layer representation.
3. Right-click the WorkOrder entity and select Add ➤ Scalar Property. Name the new property **IsPriority**. Select the IsPriority property and change its type to Boolean.
4. Right-click the .edmx file in the Solution Explorer and select Open With ➤ XML Editor.
5. Change the EntitySetMapping in the mapping layer to conditionally map the WorkOrder entity to the WorkOrder or PriorityWorkOrder table, as shown in Listing 15-26.

Listing 15-26. Changes for the .edmx file to conditionally map the WorkOrder entity to either the WorkOrder table or the PriorityWorkOrder table

```
<EntitySetMapping Name="WorkOrders">
  <EntityTypeMapping TypeName="EFRecipesModel.WorkOrder">
    <MappingFragment StoreEntitySet="WorkOrder">
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
      <ScalarProperty Name="WorkOrderId" ColumnName="WorkOrderId" />
      <ScalarProperty Name="RequestDate" ColumnName="RequestDate" />
      <ScalarProperty Name="Problem" ColumnName="Problem" />
      <Condition Name="IsPriority" Value="false" />
    </MappingFragment>
    <MappingFragment StoreEntitySet="PriorityWorkOrder">
      <ScalarProperty Name="WorkOrderId" ColumnName="WorkOrderId" />
      <ScalarProperty Name="RequestDate" ColumnName="RequestDate" />
      <ScalarProperty Name="Problem" ColumnName="Problem" />
      <Condition Name="IsPriority" Value="true" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

How It Works

To store the priority work orders in the PriorityWorkOrder table and the normal work orders in the WorkOrder table, we added the IsPriority property to the entity and used it to conditionally map the entity to the appropriate table.

Normally, each property must be mapped to a column in a table. The exception to this rule is when the property participates in a condition, as IsPriority does, and the property cannot be null, which is also true for the IsPriority property.

There is one important yet subtle problem. We have two tables that map to the same entity. This means that we have two tables mapping into a single entity set. We need to guarantee that the WorkOrderId values from the two tables never collide. These values must be unique in the entity set, which means they must be unique across both tables. There are several strategies for managing this. One approach is to assign GUIDs to the WorkOrderId. The approach we take here is to set the WorkOrderId as an integer identity column with the WorkOrder table enumerating odd integers and the PriorityWorkOrder table enumerating even integers. This effectively guarantees that the WorkOrderId will be unique across the tables.

The code in Listing 15-27 demonstrates inserting into and retrieving from our model. After running this code, check the content of the WorkOrder and PriorityWorkOrder tables. The WorkOrder table should contain the normal work orders while the PriorityWorkOrder table contains the high priority work orders.

Listing 15-27. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var wo1 = new WorkOrder { RequestDate = DateTime.Parse("11/04/09"),
                              Problem = "Printer needs paper in shipping.",
                              IsPriority = false };
    var wo2 = new WorkOrder { RequestDate = DateTime.Parse("11/04/09"),
                              Problem = "Main site database server is down!",
                              IsPriority = true };
    var wo3 = new WorkOrder { RequestDate = DateTime.Parse("11/04/09"),
                              Problem = "Backup job complete, remove tape.",
                              IsPriority = false };
    context.WorkOrders.AddObject(wo1);
    context.WorkOrders.AddObject(wo2);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    context.WorkOrders.AddObject(wo3);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("Work Orders");
    Console.WriteLine("===========");
    foreach (var wo in context.WorkOrders)
    {
        Console.WriteLine("{0}\t{1}\t{2}", wo.RequestDate.ToShortDateString(),
                          wo.Problem, wo.IsPriority ? "High" : "Normal");
    }
}
```

The output of the code in Listing 15-27 is the following:

```
Work Orders

===========

11/4/2009       Main site database server is down!      High

11/4/2009       Printer needs paper in shipping.        Normal

11/4/2009       Backup job complete, remove tape.       Normal
```

15-11. Mapping an Entity to Multiple Entity Sets (MEST)

Problem

You want to map an entity to two different entity sets, which allows you to have two different views of the same type without using inheritance.

Solution

Leveraging Multiple Entity Sets per Type, also known as MEST, allows you to map an entity into two or more entity sets each with their own associations to the entity.

Let's say we have a data model like the one shown in Figure 15-27.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

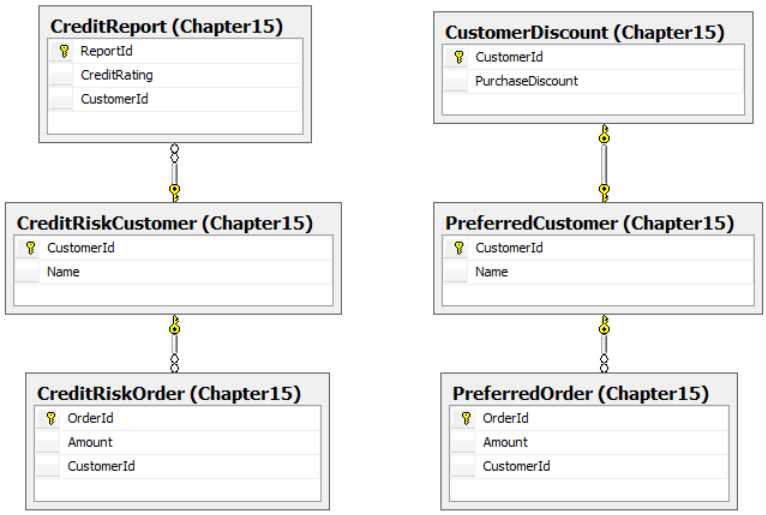


Figure 15-27. *Tables and relationships for preferred and risky customers*

The tables in Figure 15-27 depict risky and preferred customers along with separate tables for their orders. Risky customers have a relationship to their credit report information. Preferred customers have a relationship to their discount information.

We want to expose just one Customer entity at the conceptual level. This Customer entity will span two entity sets: one for the risky customers with the association to the credit reports, and one for the preferred customers with the association to the customer discounts.

In a similar vein, we want to expose just one Order entity at the conceptual level. The risky orders will live in one entity set while the preferred orders will live in a separate entity set.

To create the model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the tables shown in Figure 15-27. Or update an existing model with these tables.
2. Right-click the design surface and select Add ➤ Entity. Name the new entity **Customer**. Uncheck Create key property box.
3. Cut the CustomerId and Name properties from the CreditRiskCustomer entity and paste them into the new Customer entity.
4. Right-click the CreditRiskCustomer entity and select Delete. When prompted to delete the underlying store model tables, select No. This will delete the entity from the conceptual layer but leave it in the store layer.
5. Right-click the PreferredCustomer entity and select Delete. As in Step 4, do not delete the underlying store layer representation.
6. Right-click the design surface and select Add ➤ Entity. Name the new entity **Order**. Uncheck the Create key property box.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

7. Cut the OrderId and Amount from the CreditRiskOrder entity and paste them into the new Order entity.
8. Right-click the CreditRiskOrder entity and select delete. As in Steps 4 and 5, do not delete the underlying store model tables.
9. Right-click the PreferredOrder entity and select Delete. As in the previous steps, do not delete the underlying store layer representation.
10. Right-click the design surface and select Add ➤ Association. Select the CustomerDiscount and Customer entities for the association. Name the new association **CustomerCustomerDiscount**. Set the multiplicity to one on both sides of the association. Uncheck "Add foreign key properties to 'CustomerDiscount' entity" check box.
11. Select the new association and view its properties. Click the Referential Constraints box. In the dialog box, select Customer as the Principal. Set CustomerId as the dependent property.
12. Right-click the design surface and select Add ➤ Association. Select the CreditReport and Customer entities for the association. Name the new association **CustomerCreditReport**. Set the multiplicity to one on the Customer side and many on the CreditReport side. Uncheck "Add foreign key properties to 'CreditReport' entity" check box.
13. Select the new association and view its properties. Click the Referential Constraints box. In the dialog box, select Customer as the Principal.Set CustomerId as the dependent property.
14. Right-click the design surface and select Add ➤ Association. Select the Order and Customer entities for the association. Name the new association **RiskyCustomerRiskyOrder**. Set the multiplicity to one on the Customer side and many on the Order side. Change the name of the navigation property on the Customer side to **RiskyOrders**. Change the name of the navigation property on the Order side to **RiskyCustomer**. Uncheck the "Add foreign key properties to the 'Order' entity" check box.
15. Select the new association and view the Mapping Details window. In the Add a Table or View drop-down, select the CreditRiskOrder table. The CustomerId property should map to the CustomerId column in the Customer table. The OrderId property should map to the OrderId column in the Order table.
16. Right-click the design surface and select Add ➤ Association. Select the Order and Customer entities for the association. Name the new association **PreferredCustomerPreferredOrder**. Set the multiplicity to one on the Customer side and many on the Order side. Change the name of the navigation property on the Customer side to **PreferredOrders**. Change the name of the navigation property on the Order side to **PreferredCustomer**. Uncheck the "Add foreign key properties to the 'Order' entity" check box.
17. Select the new association and view the Mapping Details window. In the Add a Table or View drop-down, select the PreferredOrder table. The CustomerId property should map to the CustomerId column in the Customer table. The OrderId property should map to the OrderId column in the Order table.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

18. Right-click the .edmx file and select Open With ➤ XML Editor. The Customer and Order entities are now in two different entity sets. In the CSDL layer, edit the code inside **<EntityContainer>** tags using the code in Listing 15-28.

19. The associations we create must now be fixed up with the correct entity sets. In the mapping layer, make the changes shown in Listing 15-29.

After editing the .edmx, the designer will no longer open the file. The completed conceptual model is shown in Figure 15-28.

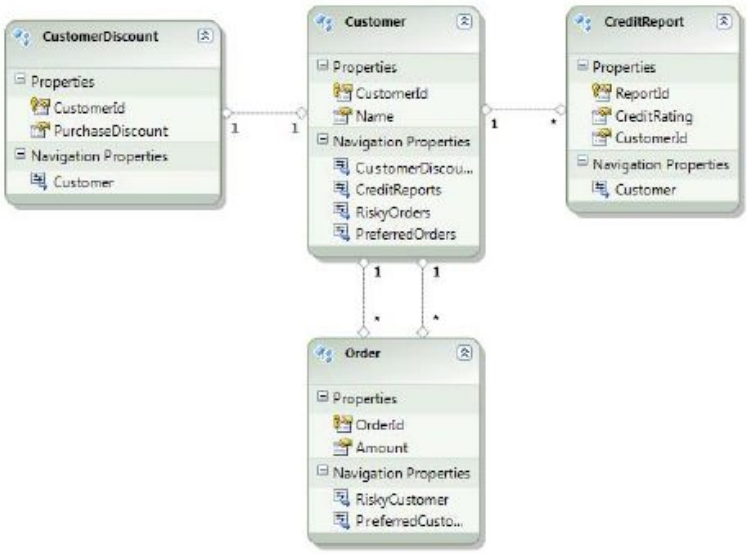


Figure 15-28. *The completed model*

Listing 15-28. Customer and Order are now each in two entity sets. Moving these entities into the entity sets also affects the mappings.

```
<EntityContainer Name="EFRecipesEntities">
  <EntitySet Name="PreferredCustomers" EntityType="EFRecipesModel.Customer" />
  <EntitySet Name="RiskyCustomers" EntityType="EFRecipesModel.Customer" />
  <EntitySet Name="PreferredOrders" EntityType="EFRecipesModel.Order" />
  <EntitySet Name="RiskyOrders" EntityType="EFRecipesModel.Order" />

  <EntitySet Name="CreditReports" EntityType="EFRecipesModel.CreditReport" />
  <EntitySet Name="CustomerDiscounts"
             EntityType="EFRecipesModel.CustomerDiscount" />

  <AssociationSet Name="CustomerCustomerDiscount"
                  Association="EFRecipesModel.CustomerCustomerDiscount">
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    <End Role="Customer" EntitySet="PreferredCustomers" />
    <End Role="CustomerDiscount" EntitySet="CustomerDiscounts" />
  </AssociationSet>

  <AssociationSet Name="CustomerCreditReport"
                  Association="EFRecipesModel.CustomerCreditReport">
    <End Role="Customer" EntitySet="RiskyCustomers" />
    <End Role="CreditReport" EntitySet="CreditReports" />
  </AssociationSet>

  <AssociationSet Name="RiskyCustomerRiskyOrder"
                  Association="EFRecipesModel.RiskyCustomerRiskyOrder">
    <End Role="Order" EntitySet="RiskyOrders" />
    <End Role="Customer" EntitySet="RiskyCustomers" />
  </AssociationSet>

  <AssociationSet Name="PreferredCustomerPreferredOrder"
                  Association="EFRecipesModel.PreferredCustomerPreferredOrder">
    <End Role="Order" EntitySet="PreferredOrders" />
    <End Role="Customer" EntitySet="PreferredCustomers" />
  </AssociationSet>
</EntityContainer>
```

Listing 15-29. Mapping our conception layer entity sets to store entity sets (we need to add only the four mappings shown here)

```
<EntitySetMapping Name="PreferredCustomers">
  <EntityTypeMapping TypeName="EFRecipesModel.Customer">
    <MappingFragment StoreEntitySet="PreferredCustomer">
      <ScalarProperty Name="CustomerId" ColumnName="CustomerId"/>
      <ScalarProperty Name="Name" ColumnName="Name"/>
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>

<EntitySetMapping Name="RiskyCustomers">
  <EntityTypeMapping TypeName="EFRecipesModel.Customer">
    <MappingFragment StoreEntitySet="CreditRiskCustomer">
      <ScalarProperty Name="CustomerId" ColumnName="CustomerId"/>
      <ScalarProperty Name="Name" ColumnName="Name"/>
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>

<EntitySetMapping Name="RiskyOrders">
  <EntityTypeMapping TypeName="EFRecipesModel.Order">
    <MappingFragment StoreEntitySet="CreditRiskOrder">
      <ScalarProperty Name="OrderId" ColumnName="OrderId" />
      <ScalarProperty Name="Amount" ColumnName="Amount" />
    </MappingFragment>
  </EntityTypeMapping>
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
</EntitySetMapping>

<EntitySetMapping Name="PreferredOrders">
  <EntityTypeMapping TypeName="EFRecipesModel.Order">
    <MappingFragment StoreEntitySet="PreferredOrder">
      <ScalarProperty Name="OrderId" ColumnName="OrderId" />
      <ScalarProperty Name="Amount" ColumnName="Amount" />
    </MappingFragment>
  </EntityTypeMapping>
</EntitySetMapping>
```

How It Works

Multiple Entity Sets per Type, often referred to simply as MEST, is a modeling approach that allows us to map a single conceptual entity to multiple entity sets. Although MEST is not a common modeling technique, we have demonstrated its use here to map an entity to two underlying database tables. In fact, we used MEST for both the Customer entity and the Order entity. This results in a clean conceptual model with a little more complex storage level model.

Associations are also first-class objects and are tied to specific entity sets. At first glance, the one-to-one association between Customer and CustomerDiscount in Figure 15-28 may look completely wrong. It seems to require every Customer to have a CustomerDiscount, even though we know that risky customers do not get discounts. Unfortunately, the design surface does not distinguish associations that live in different entity sets. The association between the PreferredCustomer and the CustomerDiscount is in an entity set that makes this one-to-one association completely valid.

When an entity is part of two or more entity sets, two or more "add" methods are generated. In our example, both **AddToPreferredCustomers()** and **AddToRiskyCustomers()** were generated by Entity Framework. The companion **PreferredCustomers.AddObject()** and **RiskyCustomers.AddObject()** methods were also generated.

As you can see from the amount of .edmx file editing required in this recipe, extensive use of MEST rapidly becomes impractical. Not the least of the impractical parts is that the changes are not supported by the designer and it will not open the .edmx file after the changes have been made.

The code in Listing 15-30 demonstrates inserting into and retrieving from our model.

Listing 15-30. Inserting and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var pc = new Customer { Name = "Steven James" };
    var rc = new Customer { Name = "Kathy Naudot" };
    pc.PreferredOrders.Add(new Order { Amount = 19.95M });
    pc.CustomerDiscount = new CustomerDiscount { PurchaseDiscount = 10 };
    rc.RiskyOrders.Add(new Order { Amount = 29.99M });
    rc.CreditReports.Add(new CreditReport { CreditRating = 630 });
    context.PreferredCustomers.AddObject(pc);
    context.RiskyCustomers.AddObject(rc);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
    Console.WriteLine("Preferred Customers");
    context.ContextOptions.LazyLoadingEnabled = true;
    foreach (var customer in context.PreferredCustomers)
    {
        Console.WriteLine("Name: {0}, Discount = {1}%", customer.Name,
                  customer.CustomerDiscount.PurchaseDiscount.ToString());
        foreach (var order in customer.PreferredOrders)
        {
            Console.WriteLine("\tOrder: {0}", order.Amount.ToString("C"));
        }
    }
    Console.WriteLine("\nRisky Customers");
    foreach (var customer in context.RiskyCustomers)
    {
        Console.WriteLine("Name: {0}", customer.Name);
        foreach (var order in customer.RiskyOrders)
        {
            Console.WriteLine("\tOrder: {0}", order.Amount.ToString("C"));
        }
        foreach (var report in customer.CreditReports)
        {
            Console.WriteLine("\tCredit Score: {0}",
                              report.CreditRating.ToString());
        }
    }
}
```

The output of the code in Listing 15-30 is the following:

```
Preferred Customers

Name: Steven James, Discount = 10%

        Order: $19.95



Risky Customers

Name: Kathy Naudot

        Order: $29.99

        Credit Score: 630
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

15-12. Extending Table per Type with Table per Hierarchy

Problem

You want to extend a Table per Type inheritance model with a Table per Hierarchy approach.

Solution

Let's say you have the tables shown in Figure 15-29.

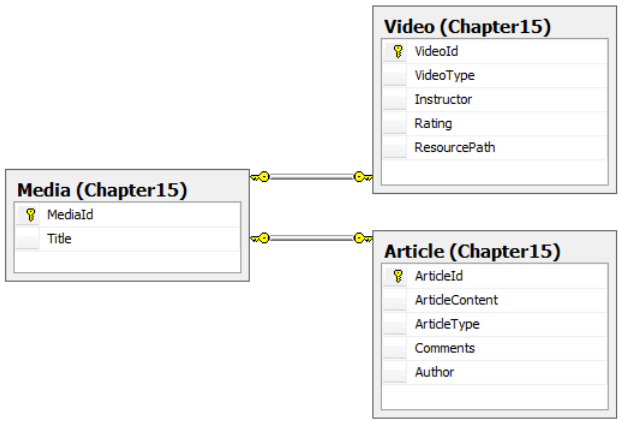


Figure 15-29. *The tables containing information about the articles and videos*

We want to create a model with an abstract base type for Media with two derived abstract types for articles and videos. We want to extend this Table per Type model by deriving new types representing specific kinds of articles and videos. In particular, we want to model the new types BlogPosting and Story by deriving from the Article base class. Also, we want to model the new types RecreationalVideo and EducationalVideo by deriving from the Video base class.

To create the Entity Data Model, do the following:

1. Add a new ADO.NET Entity Data Model to your project and import the Media, Article and Video tables. Or update an existing model with these tables.
2. Right-click the Medium entity and select Properties. Change the Abstract property to True and change the name of the entity to **Media**. Also change the entity set name to **Media**.
3. Delete the associations between the Media entity and the Article and Video entities.
4. Right-click the Video entity and select Add ➤ Inheritance. Select Media as the base entity and Video as the derived entity.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

5. Right-click the Article entity and select Add ➤ Inheritance. Select Media as the base entity and Article as the derived entity.
6. Delete the ArticleId property from the Article entity. Delete the VideoId property from the Video entity.
7. Select the Video entity and view the Mapping Details window. Map the VideoId column to the MediaId property.
8. Select the Article entity and view the Mapping Details window. Map the ArticleId column to the MediaId property.
9. Right-click the Video entity and select Properties. Set the Abstract property to True. Similarly, set the Abstract property for Article entity to True.
10. Delete the VideoType and ArticleType properties from the Video and Article entities. We will use these columns as discriminators for the Table per Hierarchy inheritance.
11. Right-click the design surface and select Add ➤ Entity. Name the new entity **BlogPosting** and uncheck the Create key property check box.
12. Right-click the design surface and select Add ➤ Entity. Name the new entity Story and uncheck the Create key property check box.
13. Right-click the BlogPosting entity and select Add ➤ Inheritance. Select Article as the base entity and BlogPosting as the derived entity. Repeat this process for the Story entity.
14. Cut the ArticleContent and Comments properties from the Article entity and paste them into the BlogPosting entity. Rename the ArticleContent property to **Post**.
15. Select the BlogPosting entity and view the Mapping Details window. In the Add a Table or View drop-down, select Article. Add the condition where `ArticleType = BlogPosting`. Map the Post property to the ArticleContent column.
16. Right-click the Story entity and select Add ➤ Scalar Property. Name the new property **Plot**.
17. Select the Story entity and view the Mapping Details. In the Add a Table or View drop-down, select Article. Add the condition where ArticleType = Story. Map the Plot property to the ArticleContent column.
18. Right-click the design surface and select Add ➤ Entity. Name the new entity **RecreactionalVideo** and uncheck the Create key property check box.
19. Right-click the design surface and select Add ➤ Entity. Name the new entity **EducationalVideo** and uncheck the Create key property checkbox.
20. Right-click the RecreationalVideo entity and select Add ➤ Inheritance. Select Video as the base entity and RecreationalVideo as the derived entity. Repeat this for the EducationalVideo entity.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

21. Cut the Rating property from the Video entity and paste it into the RecreationalVideo entity. Similarly, cut the Instructor property from the Video entity and paste it into the EducationalVideo entity.
22. Select the RecreationalVideo entity and view the Mapping Details window. In the Add a Table or View drop-down, select the Video table. Add the condition where `VideoType = RecreationalVideo`. Map the Rating column to the Rating property.
23. Select the EducationalVideo entity and view the Mapping Details window. In the Add a Table or View drop-down, select the Video table. Add the condition where `VideoType = EducationalVideo`. Map the Instructor column to the Instructor property.

The completed model is shown in Figure 15-30.

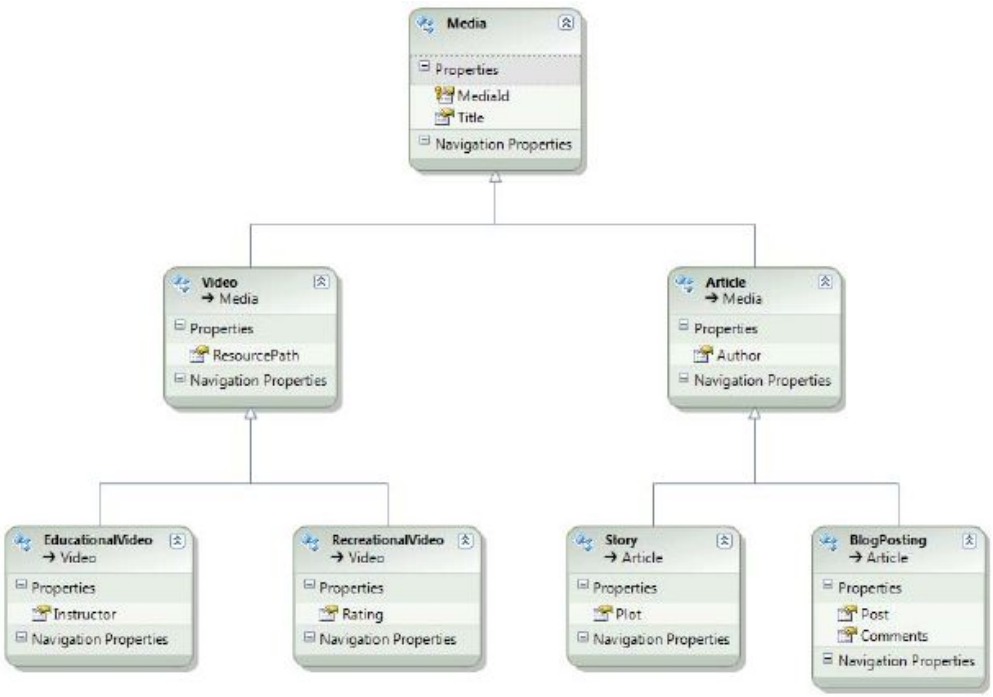


Figure 15-30. *The completed model with Table per Type inheritance extended with Table per Hierarchy inheritance*

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

How It Works

Entity Framework supports both Table per Type inheritance and Table per Hierarchy inheritance. These inheritance techniques can be combined to create an elegant and usable model.

In this recipe, we started off with Table per Type inheritance by deriving the Video and Article entities from the Media entity. This closely models the representation in the database. Next, we introduced Table per Hierarchy inheritance by synthesizing four new entities representing the types of videos and articles in our database. These new entities derived from the Article and Video entities.

The code in Listing 15-31 demonstrates inserting into and retrieving from our model.

Listing 15-31. Inserting into and retrieving from our model

```
using (var context = new EFRecipesEntities())
{
    var blogpost = new BlogPosting { Title = "ASP.NET MVC",
                            Author = "Steven Grace", Post = "What's New",
                            Comments = "50" };
    var story = new Story { Title = "Time in a Bottle",
                            Author = "Emily Jones",
                            Plot = "Murder on the high seas" };
    var ed = new EducationalVideo { Instructor = "Joseph Robins",
                            ResourcePath = "\\videos\asp.wmv",
                            Title = "ASP.NET Examples" };
    var movie = new RecreationalVideo { Title = "Archie's Place",
                            Rating = 1, ResourcePath = "\\videos\archie.wmv" };
    context.Media.AddObject(blogpost);
    context.Media.AddObject(story);
    context.Media.AddObject(ed);
    context.Media.AddObject(movie);
    context.SaveChanges();
}

using (var context = new EFRecipesEntities())
{
    Console.WriteLine("All of the media...");
    foreach(var m in context.Media)
    {
        Console.WriteLine();
        if (m is BlogPosting)
        {
            var post = (BlogPosting) m;
            Console.WriteLine("Blog Posting");
            Console.WriteLine("Title: {0}, Author: {1}, Post: {2}",
                              post.Title,post.Author,post.Post);
        }
        else if (m is Story)
        {
            var story = (Story)m;
            Console.WriteLine("Story");
            Console.WriteLine("Title: {0}, Author: {1}, Plot: {2}",
                              story.Title, story.Author, story.Plot);
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

```
        }
        else if (m is EducationalVideo)
        {
            var edvideo = (EducationalVideo)m;
            Console.WriteLine("Educational Video");
            Console.WriteLine("Title: {0}, Instructor: {1}",
                              edvideo.Title, edvideo.Instructor);
        }
        else if (m is RecreationalVideo)
        {
            var video = (RecreationalVideo)m;
            Console.WriteLine("Recreational Video");
            Console.WriteLine("Title: {0}, Rating: {1}", video.Title,
                              video.Rating);
        }
    }
}
```

The output of the code in Listing 15-31 is the following:

```
All of the media...


Blog Posting

Title: ASP.NET MVC, Author: Steven Grace, Post: What's New


Story

Title: Time in a Bottle, Author: Emily Jones, Plot: Murder on the high seas


Educational Video

Title: ASP.NET Examples, Instructor: Joseph Robins


Recreational Video

Title: Archie's Place, Rating: 1
```


Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Index

A

abstract base entities, 54
AcceptAllChanges(), 432
AcceptChanges(), 357
AccountId foreign key, 531
AddObject(), 45, 87, 268–269
AddToOrders(), 28
AddToPreferredCustomers(), 583
AddToRiskyCustomers(), 583
ADO.NET
  Entity Data Model, 5, 9, 17
  SqlClient, 65, 70
Age(), 404
@Amount parameter, 65
Any(), 77
AnyElement operator, 397
App.config file, 258
AppDomain, 276
Append(), 32
AppendOnly option, 179
Application_Start(), 139, 142
ApplicationException, 453
ApplyChanges(), 333, 338, 344
ApplyCurrentValues(), 291, 322
ApplyOriginalValues(), 322
as operator, 245
ASP.NET
  Application_Start(), 139, 142
  Attach(), 148
  Bind(), 124, 148
  building a search query using an EntityDataSource control, code listing, 115
  building CRUD operations with an ObjectDataSource control, 143
  building CRUD operations, code listing, 119
  ChangeObjectState(), 149
  ChangeObjectStatus(), 148
  ConnectionString attribute, 124
  Create(), 150, 153
  CreateDate property, 125–126
  CustomExpression, 136
  DataPager control, 119, 124
  DataSourceID attribute, 124, 135
  Delete(), 149–150, 153
  Details(), 150
  Edit(), 150, 153
  EditItemTemplate, 124
  EditTemplate, 148
  EntityDataSource control, 115, 119, 124–125, 127, 129–130, 135, 137, 139
  EntitySetName attribute, 124
  EntityTypeFilter attribute, 137
  executing business logic when changes are saved, code listing, 124
  filtering with URL routing, code listing, 139
  GetReservations(), 148
  Global.asax, 139
  GridView control, 135, 142
  HotelRepository and ReservationRepository classes, code listing, 143
  Index(), 149
  InsertItemTemplate, 124
  InsertTemplate, 148
  IQueryable<Product>, 136
  ItemTemplate, 124
  LayoutTemplate, 148
  ListView control, 119, 124, 148
  loading related entities, code listing, 127
  ObjectDataSource control, 115, 143, 145, 148
  OnContextCreated(), 126
  OnQueryCreated attribute, 142
  OrderByExpression, 136
  Page_Load event, 128
  Page_Load(), 116, 118, 124, 139
  PagedControlID, 124
  ProductsWithCategory(), 136
  ProductWithSalesGreaterThan(), 136
  PropertyExpression, 136
  QueryExtender control, 115, 130, 136, 142
  RangeExpression, 136

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

ASP.NET (*cont.*)
retrieving a derived type using an EntityDataSource control, code listing, 136
SavingChanges event, 125–126
searching with a QueryExtender control, code listing, 129
TargetControlID, 136
TotalSales property, 135
Update(), 148
using Entity Framework with MVC, procedure for, 149
Association Type relationship, definition of, 31
AssociationChanged event, code listing, 437
associations, 3
<AssociationSetMapping> tag, 229
Attach(), 87, 93, 148, 169, 172, 269, 285, 314, 318, 322, 328, 440, 476
AverageUnitPrice()
code listing, 394
using in a LINQ query, 396
using in an Entity SQL query, 396
Avg(), 419

B

BinaryFormatter, 350
Bind(), 124, 148
bitwise AND operator, 111
BitWiseAnd(), 111
BookRepository class, code listing, 306
BookRepositoryTest class, code listing, 307
bootstrapping, rules for, 400
Build Action property, 257
ByteArraySerializer class, code listing, 350

C

canonical functions
Avg(), 419
Count(), 419
definition of, 418
EntityFunction class, 421
Sum(), 419
See also custom functions; functions; model-defined functions
CartItems collection, 439
cascade delete rules
adding manually, 443
defining in both the database and the model, 443
setting, 441
using to delete related objects, code listing, 441
Changed event, 460
ChangeObjectState(), 149, 269
ChangeObjectStatus(), 148
change-tracking proxies, creating, 288
Choose Your Database Object dialog box, 23
Cleanup(), 328
Clear(), 178, 440
CLR namespace, 74
Code Generation Strategy property, 253, 276
CollectionChangedEventArgs, 440
<CollectionType> tag, 412
Common Table Expression, using in a stored procedure, 205
complex types
creating, modifying, and mapping complex types, code listing, 60
creating, modifying, and mapping complex types, procedure for, 57
definition of, 3
requirements of, 59
composite key, 264
conceptual model, 2, 9
Conceptual Schema Definition Language (CSDL), 2
concrete base entities, 54
concurrency
applying optimistic concurrency, procedure for, 509
Concurrency Mode property, 511
creating nested TransactionScopes, code listing, 517
DeleteAgent(), 516
ensuring that a forum moderator's changes overwrite a user's posts, 518
ExecuteNonQuery(), 516
ExecuteStoreCommand(), 511, 516
generating a Timestamp column with Model First, procedure for, 527
getting affected rows from a stored procedure, code listing, 521
implementing the last record wins strategy, code listing, 518
InsertAgent(), 516
inserting and updating a database using stored procedures, code listing, 514

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

managing concurrency when using stored procedures, 512
MergeOption, 520
optimistic concurrency, enabling, 511
OptimisticConcurrencyException, 520
PerserveChanges, 520
reading uncommitted data using LINQ to entities, 516
Refresh(), 512, 520
rogue updates, 524, 527
SaveChanges(), 511–512, 520
stored procedures for the Insert, Update, and Delete actions, code listing, 513
stored procedures mapped to the Insert, Update, and Delete actions, code listing, 523
StoreGeneratedPattern property, 511
throwing an exception if optimistic concurrency is violated, code listing, 510
TimeStamp property, 511, 513, 520, 524
TransactionScope class, 516
UpdateAccount(), 524
UpdateAgent(), 516
using optimistic concurrency with Table per Type inheritance, procedure for, 524, 526
See also optimistic concurrency
Concurrency Mode property, 511
conditional mappings
applying, 48
Table per Hierarchy inheritance, 48
.config file, 252
connection string, building dynamically, 251
ConnectionString attribute, 124
ConnectionStringManager class, 252
Contains clause, 417
ContextOptions, 280
Count property, 26
Count(), 77, 411, 419
Create Customer button, 342
Create(), 150, 153
CreateContext(), 256
CreateDatabase(), 309
CreateDate property, 125–126
CreateEntityKey(), 291
CreateObject(), 484
CreateObject<T>(), 289
CreateProxyTypes(), 353, 500, 502
CreateQuery(), 70, 79, 87
CreateQuery<Invoice>(), 400
CreateQuery<T>(), 404
createref(), 541
CreateSourceQuery(), 90, 92, 168, 169–171, 174, 176, 444
CreateWorkSpace(), 257
cross operator, 105
custom functions
contrasting custom functions with model-defined functions, 372
when to use, 372
See also canonical functions; functions; model-defined functions
Custom Tool Namespace property, changing, 74
CustomerRepository class, code listing, 339
CustomExpression, 136

D

database connections
accessing the CurrentState and OriginalState of the connection, 437
clicking Test Connection, 17
creating a log entry when a connection is opened or closed, 435
creating a new connection, 17
Database First, 5, 7
Database Schema Name, 11
databases
evolution of, 1
impedance mismatch between code and data, 1
DataContractResolver, 349
DataContractSerializer, 349–350, 353
DataPager control, 119, 124
DataSourceID attribute, 124, 135
DbDataReader, 70
DbDataRecord, 71
Default Value attribute, 451
DefaultIfEmpty(), 95, 105
degree, definition of, 30
Delete(), 149–150, 153
DeleteAgent(), 516
DeleteObject(), 87, 267, 285, 314, 328
DeletePayment(), 314
DeleteRelatedEntities<>(), code listing, 444

Licensed to Matt Allen 217520

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 9781430227031
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

designer
bidirectional model development, support for, 14
lack of support for model-defined functions, 396
roundtrip modeling, 14
Details(), 150
DetailsView control, 322
DetectChanges(), 292, 295, 344, 485
DetectChangesBeforeSave, 432
DiffDays(), 420–421
direction, definition of, 31
Dispose(), 16, 328
Distinct(), 89
Domain Driven Development, 271
DropDatabase(), 309

E

eager loading
code listing, 156
definition of, 155
Edit(), 150, 153
EditItemTemplate, 124
EditTemplate, 148
EdmFunction(), 412
edmgen.exe
automating some of the build processes, 264
command line options, 263
generating a model from the command line, 263
EdmGen.exe utility, generating views for a model, 479
.edmx file, 253
Build Action property, 257
contents of, 257
<FunctionImportMapping> tag, 362
EFRecipesEntities, 14, 72, 252, 257
EFRecipesModel, 72
entities
abstract base entities, 54
AddObject(), 268–269
applying an aggregate operator on related entities without loading, code listing, 173
applying conditions on a base entity, procedure for, 242
assigning default values to the properties of an entity, 447
Attach(), 169, 172, 269
automatically deleting related entities, 440
casting to ObjectQuery<T> and invoking Include(), code listing, 165
ChangeObjectState(), 269
Clear(), 178
concrete base entities, 54
CreateSourceQuery(), 168–171, 174, 176
creating a model with one entity, 9
deferred loading of related entities, 155, 167
dependent and principal entities, deleting, 265
differences between foreign key associations and independent associations, table of, 187
eager loading, code listing, 156
eager loading, definition of, 155
entity collection, 156
entity reference, 156
entity, definition of, 3
EntityKey, definition of, 3
EntitySet, definition of, 3
EntityType, definition of, 3
executing aggregate operations on related entities, 172
filtering an eagerly loaded entity collection, code listing, 180
filtering and ordering an entity collection, code listing, 170
filtering and ordering related entities, 169
First(), 165
Include(), 156, 158, 160, 163, 165, 167, 176
inserting new entities using an object context, code listing, 267
IQueryable<T>, 166
IsLoaded property, using, 174, 176
Load(), 165, 168, 173, 176–177
loading a complete object graph, 160
loading navigation properties on derived types, 162
loading related entities explicitly, 176
loading related entities in a single round trip to the database, 155
loading related entities using POCO, 276
mapping an entity based on conditions, 541
mapping an entity to customized parts of one or more tables, code listing, 534
mapping an entity to customized parts of one or more tables, procedure for, 532
mapping an entity to multiple tables, code listing, 577
mapping an entity to multiple tables, procedure for, 576

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

mapping an entity type to a subset of table rows, 46
mapping Null conditions in derived entities, code listing, 209
mapping Null conditions in derived entities, procedure for, 208
modeling Is-a and Has-a relationships between two entities, code listing, 56
modeling Is-a and Has-a relationships between two entities, procedure for, 54
modifying foreign key associations, code listing, 185
Multiple EntitySets per Type (MEST), 3
ObjectQuery<T>, 165–166
OfType<>(), 164, 172
populating entities in a Table per Hierarchy inheritance model, 376
populating entities in a Table per Type inheritance model, 373
PrintDetails(), 182
query path, definition of, 162
relationship span, 178, 182, 184
retrieving an entire object graph in a single query, code listing, 160
retrieving entities from the object state manager, 261
retrieving related entities in one round trip to the database, code listing, 163
retrieving related entities using two slightly different approaches, code listing, 168
returning an entity collection from a stored procedure, 359
SaveChanges(), 181, 268–269
sharing audit fields across multiple entities, code listing, 551
sharing audit fields across multiple entities, procedure for, 547
splitting a table across multiple entities, code listing, 40
splitting a table across multiple entities, procedure for, 37
Sum(), 174
testing whether an entity reference or entity collection is loaded, 174
ToList(), 182, 184
validating entities on the SavingChanges event, 464
where clause, 171
working with dependent entities in an identifying relationship, 264
Entity Container Name, 11
Entity Data Model Wizard, Pluralization Service, 259
Entity Framework
ADO.NET Entity Data Model, including in a project, 5
associations, 3
code-generation process, 4
coding in terms of entity types and associations, 1
ComplexType, definition of, 3
conceptual model (conceptual layer), 2
Conceptual Schema Definition Language (CSDL), 2
Database First, 7
deferred loading of related entities, 155
.edmx files, 8
Entity Data Model (EDM), 2
Entity Data Model Wizard, starting, 5
Entity SQL, 8
entity, definition of, 3
EntityClient layer, 2
EntityKey, definition of, 3
EntitySet, definition of, 3
EntityType, definition of, 3
Generate Database from Model, selecting, 7
generating a model from an existing database, 7
hints, 518
lack of support for associations with properties, 26
lack of support for lazy loading of individual entity properties, 40
lack of support for the XML data type, 460
LINQ, 8
loading only the entities directly accessed by an application, 155
Mapping Details window, 2, 7
mapping layer, 2
Mapping Specification Language (MSL), 3
Model First, 7
model, definition of, 2
modeling in, 2
Multiple EntitySets per Type (MEST), 3
navigation properties, 3
optimizing the number of database queries executed, 155
overview of, 2
POCO, 4
programming against objects in the model, 8
property, definition of, 3
scalar properties, 3
store model (store layer), 2

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Entity Framework (*cont.*)
Store Schema Definition Language (SSDL), 2
tasks, 3
testing approaches, best practices, 305
Workflow Foundation (WF), 4
working at the conceptual level for both code and data, 1
XML and models, 2
See also modeling
Entity SQL
AddObject(), 87
Attach(), 87, 93
BitWiseAnd(), 111
building and executing a query against an ObjectSet<T>, code listing, 85
calling database functions in Entity SQL, code listing, 422
CLR namespace, 74
combining the properties of two entities using a left outer join, code listing, 93
Count(), 77
CreateQuery(), 70, 79, 87
CreateSourceQuery(), 92
cross operator, 105
Custom Tool Namespace property, changing, 74
DbDataReader, 70
DbDataRecord, 71
DefaultIfEmpty(), 95, 105
DeleteObject(), 87
EFRecipesEntities, 72
EFRecipesModel, 72
EntityClient, 70
EntityDataReader, 70
ExecuteReader(), 70, 82
exists(), 77
filtering and paging a query, code listing, 98
filtering related entities, 89
finding books in a list of categories using both LINQ and Entity SQL, code listing, 82
finding the masters that have detail using both LINQ and Entity SQL, code listing, 74
flattening query results using LINQ and Entity SQL, code listing, 103
group by operator, 108
grouping by the date portion of a DateTime property, code listing, 101
grouping query results by multiple properties, code listing, 105
Include(), 92
joining two entity types on multiple properties, code listing, 111
NextResult(), 82
ObjectQuery, 70
ObjectQuery<T>, 87
ObjectSet<T>, 87
OfType(), 72
OrderBy(), 100
ordering by derived types, 96
outer apply operator, 105
Read(), 70
Recipe4 namespace, 72
retrieving a primitive type using both LINQ and Entity SQL, code listing, 87
retrieving serious accidents using CreateSourceQuery(), code listing, 90
returning multiple result sets from a stored procedure, code listing, 80
returning objects from an Entity SQL statement, code listing, 68
SaveChanges(), 70, 81
SelectValue(), 89
setting default values in a query, code listing, 77
Skip(), 100
sorting Table per Hierarchy inheritance by type, code listing, 96
specifying fully qualified names, code listing, 72
SqlClient, 70
SqlCommand, 82
SqlConnection, 82
SqlServer namespace, 424
StartsWith(), 100
Take(), 100
ToList(), 82, 92
Top(), 101
Translate(), 82
Truncate(), 102
turning off caching in, 488
using bitwise operators to filter a query, code listing, 108
using canonical functions in Entity SQL, code listing, 418
using clause, 74
Where(), 87, 100
See also LINQ; SQL
EntityClient, 2, 70
EntityCollection, 437
<EntityContainer> tag, 581

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

<EntityContainerMapping> tag, 212, 234, 533, 538
EntityDataReader, 70
EntityDataSource control, 115, 119, 124–125, 127, 129–130, 135, 137, 139
EntityFunction class, 421
EntityReference, 437
<EntitySetMapping> tag, 212, 229
EntitySetName attribute, 124
EntityTypeFilter attribute, 137
equals clause, 113
ExecuteNonQuery(), 65, 67, 516
ExecuteReader(), 70, 82
ExecuteStoreCommand(), 48–49, 64–65, 67, 511, 516, 534
ExecuteStoreQuery(), 490–491
  code listing, 67
  restrictions on using, 68
exists(), 77

F

FakeObjectSet<T>, 304
First(), 165, 474
FirstOrDefault(), 367
foreign keys
  changing an independent association into a foreign key association, procedure for, 247
  creating independent and foreign key associations, procedure for, 246
  differences between foreign key associations and independent associations, table of, 187
  Include Foreign Key Columns, 20
  limiting the values assigned to, 220
  mapping a foreign key column to multiple associations, code listing, 562
  mapping a foreign key column to multiple associations, procedure for, 556
  modifying foreign key associations, code listing, 185
  reading the foreign key constraints from a database, 18
from clause, 190, 231, 400
FullName(), 404
Function Import Wizard, 369
  mapping a custom function to a CLR method, 372
<FunctionImportMapping> tag, 362, 374–378
functions
  Age(), 404
  AnyElement operator, 397
  AverageUnitPrice(), code listing, 394
  Avg(), 419
  calling a model-defined function from a model-defined function, code listing, 406
  calling a model-defined function from a model-defined function, procedure for, 404
  calling database functions in Entity SQL, code listing, 422
  calling database functions in LINQ, code listing, 424
  canonical function, definition of, 418
  <CollectionType> tag, 412
  contrasting custom functions with model-defined functions, 372
  Count(), 411, 419
  CreateQuery<T>(), 404
  defining built-in functions, procedure for, 425
  DiffDays(), 420–421
  EdmFunction(), 412
  Entity SQL, 396
  EntityFunction class, 421
  filtering an entity collection using a model-defined function, procedure for, 397
  FullName(), 404
  GetInvoices(), code listing, 398
  GetProjectManager(), 408
  GetSupervisor(), 408
  GetVisitSummary(), code listing, 413
  ISNULL(), code listing, 426
  model-defined functions, 394, 396
  MyFunction class, 396
  native functions, 372
  overview of, 393
  PlatinumSponsors(), code listing, 416
  returning a collection of entity references from a model-defined function, procedure for, 415
  returning a complex type from a model-defined function, procedure for, 412
  returning a scalar value from a model-defined function, procedure for, 393
  returning an anonymous type from a model-defined function, procedure for, 408
  <RowType> tag, 412
  <Schema> tag, 394, 398, 401
  SqlFunctions class, 425
  SqlServer namespace, 424
  Sum(), 419

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

functions (*cont.*)
treat(), 408
using canonical functions in Entity SQL, code listing, 418
using canonical functions in LINQ, code listing, 420
VisitorSummary(), code listing, 410
when to use custom functions, 372
See also canonical functions; custom functions; model-defined functions

G

garbage collector, waiting to reclaim resources, 16
Generate Database from Model, selecting, 7
Generate Database Script from Model, 12
GetAllMedia stored procedure, 374
GetAllMedia(), code listing, 374
GetAllPeople stored procedure, 377
GetAllPeople(), code listing, 377
GetClient(), 349
GetConnection(), 252
GetCustomer(), 344
GetCustomers(), code listing, 361
GetEmployeeAddresses(), code listing, 368
GetEntities<T>(), 263
GetInvoices()
bootstrapping, rules for, 400
code listing, 398
CreateQuery<Invoice>(), 400
Include(), 400
IQueryable<Invoice>, 400
ObjectQuery<Invoice>, 400
returning a computed column from a model-defined function, code listing, 402
returning a computed column from a model-defined function, procedure for, 401
GetKnownProxyTypes(), 502
GetObjectByKey(), 289, 291, 473
GetObjectStateEntries(), 263
GetPostByTitle(), 328
GetProjectManager(), 408
GetReservations(), 148
GetSubCategories(), 205–207
GetSupervisor(), 408
GetVehiclesWithRentals(), code listing, 364
GetVisitSummary(), code listing, 413
GetWithdrawals(), code listing, 366
Global.asax, 139
GridView control, 135, 142
group by operator, 108, 165, 167

H

hints, 518

I

ICollection<T>, 279, 286, 289, 483, 485
identifying relationship
definition of, 265
DeleteObject(), 267
deleting a dependent entity, code listing, 265
dependent and principal entities, deleting, 265
working with dependent entities, 264
ImageURL scalar property, 34
impedance mismatch problem, 1
in clause, 84
Include Foreign Key Columns, 20
Include(), 92, 156, 160, 163, 176, 205, 400
invoking, 158
lack of support for a filtering predicate in Entity Framework, 180
object graph and, 158
performance drawbacks of, 497
performance implications of using, 158
query path, definition of, 162
rules for using, 167
SQL query, 158
using with other LINQ query operators, 165
Index(), 149
inheritance
derived type extending the properties of a base type, 44
inheritance models supported by Entity Framework, 46
modeling Table per Type inheritance, code listing, 44
modeling Table per Type inheritance, procedure for, 42
Table per Concrete Type, 46
Table per Hierarchy, 46
Table per Type, 46
insert statement, 516
InsertAgent(), 516
InsertItemTemplate, 124
InsertOrder(), 318

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

InsertPayment(), 314
InsertPost(), 328
InsertTemplate, 148
Int32, 10
into clause, 108
InvalidOperationException, 568
IObjectChangeTracker interface, 333
IObjectSet<T>, 304
IQueryable<Invoice>, 400
IQueryable<Product>, 136
IQueryable<T>, 166
IReservationContext interface, 297
Is Not Null condition, 54, 209, 218
Is Null condition, 48, 218
is operator, 54
IsActive property, 432, 434
IsBackOrderable property, 533
IsComposable attribute, 362
ISet<T>, 273
IsLoaded property
  code listing, 174
  using, 174, 176
ISNULL()
  code listing, 426
  defining in the store layer, 428
IsPriority property, 577
IsRelationship property, 457
ItemTemplate, 124
IValidate interface, 296
IValidator interface
  defining, 465
  Validate(), 465, 469

K

Key Property, 10

L

LayoutTemplate, 148
lazy loading, 40–41
LazyLoadingEnabled, 280
let clause, 97
let keyword, 480
link table, 23
  creating a synthetic key for, 195
  exposing as an entity, procedure for, 192
  importing a link table with no payload into a model, 555
  obtaining the underlying keys, 191
  representing a many-to-many relationship, 190
  retrieving in a many-to-many association, 189
LINQ, 1, 8
  Any(), 77
  bitwise AND operator, 111
  calling database functions in LINQ, code listing, 424
  Distinct(), 89
  finding books in a list of categories using both LINQ and Entity SQL, code listing, 82
  finding the masters that have detail using both LINQ and Entity SQL, code listing, 74
  flattening query results using LINQ and Entity SQL, code listing, 103
  LINQ to Entities, querying a model, 22
  retrieving a primitive type using both LINQ and Entity SQL, code listing, 87
  Select(), 89
  setting default values in a query, code listing, 77
  using canonical functions in LINQ, code listing, 420
  See also Entity SQL; SQL
ListView control, 119, 124, 148
Load(), 168, 173, 176, 199, 205, 207, 440
  AppendOnly option, 179
  merge options, 179
  NoTracking option, 179
  OverwriteChanges option, 179
  partial loading of an entity collection, code listing, 179
  performance implications of using, 165
  PreserveChanges option, 179
  using, code listing, 177
LoadProperty(), 279
  using to load navigation properties, code listing, 277
lookup tables, overloading, 222

M

Main(), 314, 318
many-to-many relationship, 190
Mapping Details window, 2, 7, 34
mapping layer, 2
Mapping Specification Language (MSL), 3
MarkAsAdded(), 333
MarkAsDeleted(), 333, 344
MarkAsModified(), 333, 344
MarkAsUnchanged(), 333, 344
MembersWithTheMostMessages(), code listing, 371

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

MergeOption property, 520
AppendOnly option, 82, 476
NoTracking option, 68, 475
OverwriteChanges option, 491
table of options, 87
Metadata Artifact Processing property, 253, 257–258
metadata tag, 258
MetadataWorkspace
creating, 253
definition of, 257
model-defined functions, 394
acceptable parameter types, 396
best practices for employing, 397
Entity SQL, 396
See also canonical functions; custom functions; functions
Model First, 5, 7, 527
Model.csdl file, 257
Model.msl file, 257
Model.ssdl file, 257
modeling
adding an extra integer identity column to a link table, 29
AddToPreferredCustomers(), 583
AddToRiskyCustomers(), 583
applying conditions in Table per Type inheritance, code listing, 226
applying conditions in Table per Type inheritance, procedure for, 224
applying conditions on a base entity, procedure for, 242
AssociationSet mappings with conditions, code listing, 566
<AssociationSetMapping> tag, 229
bidirectional model development, designer's support for, 14
changing an independent association into a foreign key association, procedure for, 247
conditional associations, modeling, 538
createref(), 541
creating a database script, 12
creating a filter on multiple criteria, code listing, 228
creating a filter on multiple criteria, procedure for, 227
creating a model from an existing database, 16
creating a simple conceptual model, procedure for, 9
creating an association on a derived entity, code listing, 531
creating an association on a derived entity, procedure for, 529
creating conditional associations, procedure for, 536
creating independent and foreign key associations, procedure for, 246
creating read-only and computed properties, code listing, 574
creating read-only and computed properties, procedure for, 568
creating, modifying, and mapping complex types, code listing, 60
creating, modifying, and mapping complex types, procedure for, 57
deploying a model, 257
<EntityContainer> tag, 581
<EntityContainerMapping> tag, 212, 234, 533, 538
Entity Framework's approach to, 2
entity set mappings for the Audits, code listing, 548
<EntitySetMapping> tag, 212, 229
ExecuteStoreCommand(), 534
exposing a link table as an entity, procedure for, 192
extending a Table per Type inheritance model with a Table per Hierarchy approach, code listing, 588
extending a Table per Type inheritance model with a Table per Hierarchy approach, procedure for, 585
fabricating additional inheritance hierarchies, code listing, 545
fabricating additional inheritance hierarchies, procedure for, 542
Generate Database Script from Model, 12
generating a database for a model, 11
GetSubCategories(), 205–207
handling validation in the SavingChanges event, code listing, 237
implementing a complex filter using QueryView, 227
importing a link table with no payload into a model, 555
importing the view, tables, and relationships into a model, procedure for, 17
Include Foreign Key Columns, 20
Include(), 205

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

inheritance models supported by Entity Framework, 46
Insert, Update, and Delete actions for the Friend, Relative, and Residence entities, code listing, 559
inserting and retrieving WebOrders, code listing, 231
inserting into and querying of a model, 20
inserting into and retrieving rows from the Account table, code listing, 48
inserting into and retrieving Task and Worker entities, code listing, 195
InvalidOperationException, 568
Is Not Null condition, 218
Is Null condition, 218
IsBackOrderable property, 533
IsPriority property, 577
limiting the values assigned to a foreign key, code listing, 223
limiting the values assigned to a foreign key, procedure for, 220
link table, 23
Load(), 199, 205, 207
mapping a foreign key column to multiple associations, code listing, 562
mapping a foreign key column to multiple associations, procedure for, 556
mapping an entity based on conditions, 541
mapping an entity to customized parts of one or more tables, code listing, 534
mapping an entity to customized parts of one or more tables, procedure for, 532
mapping an entity to Multiple EntitySets per Type (MEST), code listing, 583
mapping an entity to Multiple EntitySets per Type (MEST), procedure for, 578
mapping an entity to multiple tables, code listing, 577
mapping an entity to multiple tables, procedure for, 576
mapping conception layer entity sets to store entity sets, code listing, 582
mapping Null conditions in derived entities, code listing, 209
mapping Null conditions in derived entities, procedure for, 208
mapping the HireDate and Salary properties, code listing, 544
mappings for the many-to-many association using the vwAuthorBook view, 554
merging two entities into a single entity, procedure for, 34
model, definition of, 2
modeling a many-to-many relationship with a payload, code listing, 27
modeling a many-to-many relationship with a payload, procedure for, 26
modeling a many-to-many relationship with no payload, code listing, 24
modeling a many-to-many relationship with no payload, procedure for, 22
modeling a many-to-many relationship with payload, code listing, 555
modeling a many-to-many relationship with payload, procedure for, 553
modeling a many-to-many, self-referencing relationship, code listing, 197
modeling a many-to-many, self-referencing relationship, procedure for, 196
modeling a self-referencing relationship and retrieving a complete hierarchy, procedure for, 204
modeling a self-referencing relationship using Table per Hierarchy inheritance, code listing, 202
modeling a self-referencing relationship using Table per Hierarchy inheritance, procedure for, 200
modeling a self-referencing relationship, code listing, 31
modeling a self-referencing relationship, procedure for, 29
modeling Is-a and Has-a relationships between two entities, code listing, 56
modeling Is-a and Has-a relationships between two entities, procedure for, 54
modeling nested Table per Hierarchy inheritance, code listing, 218
modeling nested Table per Hierarchy inheritance, procedure for, 216
modeling Table per Concrete Type inheritance, code listing, 241
modeling Table per Concrete Type inheritance, procedure for, 238
modeling Table per Hierarchy inheritance, code listing, 52
modeling Table per Hierarchy inheritance, procedure for, 49
modeling Table per Type inheritance using a non-primary key column, procedure for, 211, 213

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

modeling (*cont.*)
modeling Table per Type inheritance, code listing, 44
modeling Table per Type inheritance, procedure for, 42
Multiple EntitySets per Type (MEST), 549
object graph, building, 22
OfType<>(), 244
PromoteToMedicine(), 210
querying an entity data model, 63
QueryView, 215, 227
QueryView and procedure mappings for the associations, code listing, 540
refactoring a model, 29
representing a many-to-many relationship as two one-to-many associations, 26
retrieving a link table, code listing, 190
retrieving the link table in a many-to-many association, 189
roundtrip modeling, 14
SaveChanges(), 211
SelectMany(), 190–191
sharing audit fields across multiple entities, code listing, 551
sharing audit fields across multiple entities, procedure for, 547
solving the entity key problem, 551
splitting a table across multiple entities, code listing, 40
splitting a table across multiple entities, procedure for, 37
splitting an entity across multiple tables, procedure for, 33
stored procedures for the Insert, Update, and Delete actions for the Customer and Order entities, code listing, 570
Stored procedures for the Insert, Update, and Delete actions for the entities, code listing, 538
Table per Hierarchy inheritance, 529, 542
ToList(), 223
using a Common Table Expression in a stored procedure, 205
using a recursive method to form the transitive closure, code listing, 198
using complex conditions with Table per Hierarchy inheritance, code listing, 236
using complex conditions with Table per Hierarchy inheritance, procedure for, 232
using conditions to filter an objectset, procedure for, 46
using inheritance to map a foreign key column to multiple associations, code listing, 567
using inheritance to map a foreign key column to multiple associations, procedure for, 564
using LINQ to Entities to query a model, 22
using the as operator, code listing, 245
Validate(), 238
vertical splitting, definition of, 35
when two tables map into a single entity set, 577
working with payload-free, many-to-many relationships, 29
working with the vertically split Product entity type, code listing, 35
See also Entity Framework
<ModificationFunctionMapping> tag, 387
Multiple EntitySets per Type (MEST), 3, 549
mapping an entity to, code listing, 583
mapping an entity to, procedure for, 578
multiplicity, definition of, 30
MyFunction class, 396

N

native functions, 372
navigation properties, 3, 19
new operator, 8
NextResult(), 82, 365
NOLOCK query hint, 518
NoTracking option, 179
n-tier applications
AcceptChanges(), 357
ApplyChanges(), 333, 338, 344
ApplyCurrentValues(), 322
ApplyOriginalValues(), 322
Attach(), 314, 318, 322, 328
BinaryFormatter, 350
BookingClient test client code, 337
ByteArraySerializer class, code listing, 350
Cleanup(), 328
Client POCO class and object context, code listing, 346
Create Customer button, 342
CreateProxyTypes(), 353
CustomerRepository class, code listing, 339
DataContractResolver, 349
DataContractSerializer, 349–350, 353
DeleteObject(), 314, 328
DeletePayment(), 314

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

deleting an entity when disconnected, procedure for, 311
DetailsView control in an ASP.NET web page, code listing, 320
DetectChanges(), 344
Dispose(), 328
EnrollmentClient test client code, 332
extending the EFRecipesEntities class, code listing, 340
finding out which properties have changed, procedure for, 319
fixing duplicate references on a WCF client, procedure for, 354
GetClient(), 349
GetCustomer(), 344
GetPostByTitle(), 328
implementation of the IService1 interface, code listing, 331, 335, 347, 356
implementation of the service contract, code listing, 325
InsertOrder(), 318
InsertPayment(), 314
InsertPost(), 328
IObjectChangeTracker interface, 333
Main(), 314, 318
managing concurrency when disconnected, procedure for, 315
MarkAsAdded(), 333
MarkAsDeleted(), 333, 344
MarkAsModified(), 333, 344
MarkAsUnchanged(), 333, 344
ObjectDataSource control, 319, 322
ObjectMaterialized event, 344
Order POCO class and related object context, code listing, 316
POCO classes serialized by the WCF service, code listing, 312
Post, Comment, and EFReceipesEntities object context, code listing, 324
project repository with UpdateProject(), code listing, 319
ProxyDataContractResolver class, 345, 349
Read Customer button, 342, 344
Repository pattern, 319
SaveChanges(), 314–315, 318, 333
Self-Tracking Entities template, 329
Serializable attribute, 353
serializing proxies in a WCF service, procedure for, 345
serializing self-tracking entities in the ViewState, procedure for, 349
service contract for the WCF service, code listing, 313
StartSelfTracking(), 339, 344
StartTracking(), 344
StopTracking(), 344
SubmitCategory(), 358
SubmitCustomerWithPhones(), 344
SubmitPost(), 328
Update Customer button, 342, 344
UpdateOrderByRetrieving(), 318
UpdateOrderWithoutRetrieving(), 318
UpdateProject(), 322
using POCO with WCF, procedure for, 323
using self-tracking entities on the server side, procedure for, 338
using self-tracking entities with WCF, procedure for, 329
validating self-tracking entities, procedure for, 334
WCF Service Library, creating, 312
Windows console application serving as a test client, code listing, 327
Null condition, 209

O

object graph, building, 22
Object Relational Mapping (ORM), 1
object services
building a connection string dynamically, code listing, 251
conceptual layer (CSDL), 253
connection string syntax for loading model layers, table of, 258
ConnectionStringManager class, 252
CreateContext(), 256
CreateWorkSpace(), 257
deploying a model, 257
.edmx file, 253
EFRecipesEntities, 257
embedding the model layers as resources in an assembly, 258
GetConnection(), 252
mapping layer (MSL), 253
Metadata Artifact Processing property, 257–258
metadata tag, 258
MetadataWorkspace, creating, 253
MetadataWorkspace, definition of, 257
Model.csdl file, 257
Model.msl file, 257

Licensed to Matt Allen 2175280

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

object services (*cont.*)
Model.ssdl file, 257
OnContextCreated(), 252
POCO, 257
reading a model from a database, procedure for, 253
reading the metadata from the Definitions table, code listing, 254
SaveChanges(), 256
storage layer (SSDL), 253
using the connection string found in the .config file, 252
XmlReaders, 257
object state manager, 288
GetEntities<T>(), 263
GetObjectStateEntries(), 263
retrieving all entities in the Added, Modified, or Unchanged state, code listing, 261
retrieving entities from, 261
SavingChanges event, 263
ObjectContext, 14
ObjectDataSource control, 115, 143, 145, 148, 319, 322
ObjectMaterialized event, 344
ObjectQuery, 70
ObjectQuery<Invoice>, 400
ObjectQuery<T>, 87, 165–166, 476
objects
applying server-generated values to properties, code listing, 462
applying server-generated values to properties, procedure for, 460
assigning default values to the properties of an entity, 447
Attach(), 440
automatically deleting related entities, 440
business rule validation and enforcement, best practices, 469
CartItems collection, 439
cascade delete rules, setting, 441
Changed event, 460
Clear(), 440
CollectionChangedEventArgs, 440
CreateSourceQuery(), 444
Default Value attribute, 451
DeleteRelatedEntities<>(), code listing, 444
deleting all related entities in a generic way, 443
enforcing the order-fulfillment stages, code listing, 454
EntityCollection, 437
EntityReference, 437
executing code when SaveChanges() is called, code listing, 430
handling the SavingChanges event to enforce the business rule, code listing, 451
handling the SavingChanges event to set the default values, code listing, 448
IsActive property, 432, 434
IsRelationship property, 457
IValidator interface, defining, 465
Load(), 440
logging database connections, code listing, 435
monitoring the changing of the UserName property, code listing, 432
OnContextCreated(), 435, 437, 469
OnUserNameChanged(), 432, 434
OnUserNameChanging(), 432, 434
PropertyChanged event, 434, 440
PropertyChanging event, 434
PropertyEventArgs parameter, 434
recalculating a property value when an entity collection changes, 437
RelatedEnd, 437
retrieving the original association for independent associations, 454
retrieving the original value of a property, 451
SaveChanges(), 469
setting a property's StoreGeneratedPattern, 464
setting the default value through the store layer, 448
StateChange event, 435, 437
StateChangeEventArgs parameter, 437
Sum(), 440
throwing an ApplicationException, 453
treating a scalar property of type string as XML data, 457
User entity, 432
using the AssociationChanged event, code listing, 437
using the CandidateResume property to expose the resume as XML, code listing, 458
using the cascade delete rules to delete related objects, code listing, 441
Validate(), 465, 469
validating entities on the SavingChanges event, 464
validating SaleOrder entities in the SavingChanges event, code listing, 465
XElement class, 458
ObjectSet<T>, 87, 274, 276, 476

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

OfType<T>(), 472
OfType<>(), 172, 244
selecting instances of a given subtype from an entity set, 164
OnContextCreated(), 126, 252, 435, 437, 469
OnQueryCreated attribute, 142
OnUserNameChanged(), 432, 434
OnUserNameChanging(), 432, 434
operators
AnyElement, 397
as, 245
bitwise AND, 111
cross, 105
group by, 165, 167
is, 54
new, 8
outer apply, 105
optimistic concurrency
Concurrency Mode property, 511
enabling, 511
procedure for applying, 509
throwing an exception if optimistic concurrency is violated, code listing, 510
using with Table per Type inheritance, procedure for, 524, 526
See also concurrency
OptimisticConcurrencyException, 520
OrderBy(), 100
OrderByExpression, 136
outer apply operator, 105
OverwriteChanges option, 179

P

Page_Load event, 128
Page_Load(), 116, 118, 124, 139
PagedControlID, 124
partial methods, 434
performance improvements
adding a QueryView for each derived type, code listing, 497
Attach(), 476
building a search query efficiently, 479
comparing the performance of a simple compiled LINQ query, code listing, 486
compiling LINQ queries, 485
composing compiled and noncompiled queries, code listing, 489
CreateObject(), 484
CreateProxyTypes(), 500, 502
DetectChanges(), 485
eagerly loading a related collection without using Include(), code listing, 495
ExecuteStoreQuery(), 490–491
first(), 474
generating proxies explicitly, 500
generating tracking proxies before loading the entities, code listing, 501
GetKnownProxyTypes(), 502
GetObjectByKey(), 473
ICollection<T>, 483, 485
improving QueryView performance, 497
improving the startup time, procedure for, 477
making change tracking with POCO faster, 482
measuring the execution time of a simple query, code listing, 478
moving an expensive property to another entity, procedure for, 491
ObjectQuery<T>, 476
ObjectSet<T>, 476
OfType<T>(), 472
optimizing queries in a Table per Type inheritance model, code listing, 471
performing a simple query using the NoTracking merge option, code listing, 475
POCO classes with properties marked as virtual, code listing, 483
preventing the update of all columns in self-tracking entities, procedure for, 503
retrieving a single entity using an entity key, code listing, 473
retrieving entities for read only, 475
returning partially filled entities, code listing, 489
SaveChanges(), 485
Service1 implementation in the Service1.svc.cs file, code listing, 505
single(), 474
SQL generated when let is used in the LINQ query, 482
TestClient console application, code listing, 506
TryGetObjectByKey(), 474
UpdateComplaint(), 504
using Compile() for queries that return an anonymous type, code listing, 488
using CompileQuery.Compile() to compile a query, 486
using the CSharp.Views.tt T4 template, 478
using the let keyword and explicit conditions in a query, code listing, 480

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

performance improvements (*cont.*)
using the ResumeDetail entity, code listing, 494
PerserveChanges, 520
PlatinumSponsors(), code listing, 416
Pluralization Service
enabling, 259
Pluralize New Objects property, 260
pluralizing and singularizing words, code listing, 260
setting the default on/off state, 260
System.Data.Entity.Design namespace, 260
using to import tables from a database, 258
POCO
AppDomain, 276
ApplyCurrentValues(), 291
Attach(), 285
automating the creation and dropping of a test database, 308
BookRepository class, code listing, 306
BookRepositoryTest class with unit tests, code listing, 307
calling DetectChanges() and handling the SavingChanges event, code listing, 292
change tracking with proxies, 485
change tracking with snapshots, 485
change-tracking proxies, creating, 288
classes and the Entity Framework, 4
Code Generation Strategy property, 276
ContextOptions, 280
CreateDatabase(), 309
CreateEntityKey(), 291
CreateObject<T>(), 289
creating a class derived from ObjectContext, code listing, 274
creating classes for the entities in the customers' orders model, code listing, 273
creating unit tests for defined business rules, 296
DeleteObject(), 285
deleting a POCO entity with a complex type, code listing, 285
DetectChanges(), 295
DropDatabase(), 309
EFRecipesEntities object context, 276
FakeObjectSet<T>, 304
GetObjectByKey(), 289, 291
ICollection<T>, 279, 286, 289
implementing an object context specific to the model and entities, 276
implementing the fake object set and fake object context, code listing, 300
IObjectSet<T>, 304
IReservationContext interface, 297
ISet<T>, 273
IValidate interface, 296
IValidate interface, implementing for the Reservation and Schedule classes, 298
lack of support for lazy loading and change tracking, 276
lack of support for struct as a complex type, 283
lazy loading of related entities, code listing, 279
LazyLoadingEnabled, 280
loading related entities using POCO, 276
making change tracking with POCO faster, 482
manually synchronizing the object graph and the object state manager, 292
marking properties as virtual, 280
notifying Entity Framework about object changes, code listing, 286
object state manager, 288
ObjectSet<T>, 274, 276
Plain Old CLR Objects, explanation of, 271
POCO template, using, 296
ReservationRepository class, 300
retrieving the original object from a database, code listing, 289
rules for using complex types with POCO, 285
SaveChanges(), 285, 295, 299
SavingChanges event, 295
testing a repository against a database, procedure for, 305
testing domain objects, procedure for, 296
unit tests for the Tests project, code listing, 303
using a complex type in a POCO entity, code listing, 283
using LoadProperty() to load navigation properties, code listing, 277
using POCO in an application, code listing, 274
using POCO in an application, procedure for, 271
PreserveChanges option, 179
primary keys, 33
Print(), 32
PrintDetails(), 182
ProductsWithCategory(), 136
ProductWebInfo entity, 34
ProductWithSalesGreaterThan(), 136
PromoteToMedicine(), 210

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

properties
ComplexType, definition of, 3
marking as virtual, 280
navigation, 3, 19
property, definition of, 3
scalar, 3, 19
Property Type, 10
PropertyChanged event, 434, 440
PropertyChanging event, 434
PropertyEventArgs parameter, 434
PropertyExpression, 136
ProxyDataContractResolver class, 345, 349

Q

query path, definition of, 162
QueryExtender control, 115, 130, 136, 142
QueryView, 215, 534
common use cases for using, 230
Entity SQL, 230
implementing a complex filter, 227

R

RangeExpression, 136
Read Customer button, 342, 344
Read(), 70
Refresh(), 512, 520
RelatedEnd, 437
relationship span
definition of, 178, 184
using, 182
RelationshipManager, 444
Repository pattern, 319
ReservationRepository class, 300
rogue updates, 524, 527
root entity, 32
roundtrip modeling, 14
<RowType> tag, 412

S

SaveChanges()
AcceptAllChanges(), 432
DetectChangesBeforeSave, 432
executing code when SaveChanges() is called, code listing, 430
overriding, 299, 431
validating property changes, 432
SavingChanges event, 125–126, 263, 295
handling in order to enforce the business rule, code listing, 451
handling in order to set the default values, code listing, 448
scalar properties, 3, 10, 19
<Schema> tag, 394, 398, 401
select statement, 230, 404, 516
Select(), 89
SelectMany(), 190–191
SelectValue(), 89
Self-Tracking Entities template, 329
Serializable attribute, 353
single(), 474
Skip(), 100
Solution Explorer, 398
SQL
@Amount parameter, 65
ExecuteNonQuery(), 65, 67
ExecuteStoreCommand(), 64–65, 67
ExecuteStoreQuery(), 67–68
executing an SQL statement, code listing, 63
injection attacks, 66
returning objects from an SQL statement, 66
SaveChanges(), 67
using parameters for SQL statements, best practices, 66
@Vendor parameter, 65
See also Entity SQL; LINQ
SQL Server Management Studio, 441
SqlClient, 65, 70
SqlCommand, 82
SqlConnection, 82
SqlFunctions class, 425
SqlServer namespace, 424
StartSelfTracking(), 339, 344
StartsWith(), 100
StartTracking(), 344
StateChange event, 435, 437
StateChangeEventArgs parameter, 437
StopTracking(), 344
store model, 2
Store Schema Definition Language (SSDL), 2
stored procedures
Add Function Import dialog box, 360
defining a custom function in the storage model, 370
definition of, 359
FirstOrDefault(), 367
Function Import Wizard, 369

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

stored procedures (*cont.*)
<FunctionImportMapping> tag, 374–375, 377–378
GetAllMedia(), code listing, 374
GetAllPeople(), code listing, 377
GetCustomers(), code listing, 361
GetEmployeeAddresses(), code listing, 368
GetSubCategories(), 205–207
getting affected rows from a stored procedure, code listing, 521
GetVehiclesWithRentals(), code listing, 364
GetWithdrawals(), code listing, 366
IsComposable attribute, 362
managing concurrency when using stored procedures, 512
mapping stored procedures to actions, best practices, 381
mapping the Insert, Update, and Delete actions to stored procedures, 379
mapping the Insert, Update, and Delete actions to stored procedures for Table per Hierarchy inheritance, code listing, 391
mapping the Insert, Update, and Delete actions to stored procedures for Table per Hierarchy inheritance, procedure for, 387
mapping the Insert, Update, and Delete actions to stored procedures, code listing, 381
MembersWithTheMostMessages(), code listing, 371
<ModificationFunctionMapping> tag, 387
NextResult(), 365
populating entities in a Table per Hierarchy inheritance model, 376
populating entities in a Table per Type inheritance model, 373
returning a complex type from a stored procedure, 367
returning a scalar value result set, 365
returning an entity collection, 359
returning output parameters, 362
Stored Procedure Mapping, 379
stored procedures for the Insert and Delete actions in a many-to-many association, 382
stored procedures for the Insert and Delete actions in a many-to-many association, code listing, 383
stored procedures for the Insert and Delete actions in a many-to-many association, SQL Profiler output, 385
ToList(), 365
StoreGeneratedPattern property, 10, 511
StringBuilder class, 32
SubmitCategory(), 358
SubmitCustomerWithPhones(), 344
SubmitPost(), 328
Sum(), 174, 419, 440

T

Table per Concrete Type inheritance, 46
ensuring a unique entity key across tables, 240
performance advantages and disadvantages of, 241
practical applications for, 241
procedure for modeling relationships in, 238
Table per Hierarchy inheritance, 46, 48, 529, 542
modeling a self-referencing relationship, code listing, 202
modeling a self-referencing relationship, procedure for, 200
modeling nested Table per Hierarchy inheritance, code listing, 218
modeling nested Table per Hierarchy inheritance, procedure for, 216
rules for using, 54
using a single table to represent an inheritance hierarchy, 52
using complex conditions with, 232
Table per Type inheritance, 46
applying conditions in, 224
modeling using a non-primary key column, procedure for, 211, 213
optimizing queries, code listing, 471
tables
giving singular or plural names to, 259
inserting a row with a non-Null value, 48
mapping an entity type to a subset of table rows, 46
People table, creating, 12
self-referencing, 31
two or more tables with common primary keys, 33
vertical splitting, definition of, 35
Take(), 100
TargetControlID, 136
tasks, 3

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.

Test Connection, 17
Text Template Transformation Toolkit (T4 Templates), 4
Self-Tracking Entities template, 507
using the CSharp.Views.tt template, 478
TimeStamp property, 511, 513, 520, 524
ToList(), 82, 92, 182, 184, 223, 365
Top(), 101
ToString(), 32
TotalSales property, 135
TransactionScope class, 516
transitive closure, 198
Translate(), 82
treat(), 408
Truncate(), 102
TryGetObjectByKey(), 474

U

Update Customer button, 342, 344
Update From Database Wizard, 372
Update Model From Database, 360
update statement, 511, 516, 520
Update(), 148
UpdateAccount(), 524
UpdateAgent(), 516
UpdateComplaint(), 504
UpdateOrderByRetrieving(), 318
UpdateOrderWithoutRetrieving(), 318
UpdateProject(), 322
User entity, 432
using clause, 74
using(), 16

V

Validate(), 238, 465, 469
@Vendor parameter, 65
vertical splitting
additional join required by, 35
definition of, 35
VisitorSummary(), code listing, 410
Visual Studio 2010
Command Prompt, accessing, 263
Database First, 5
features of, 4
importing tables and relationships into a model, 5
Model First, 5
providing an integrated design surface for Entity Framework models, 5
Text Template Transformation Toolkit (T4 Templates), 4
updating a model from its database, 5
Workflow Foundation (WF), 4
vwLibrary, 17, 22

W

web.config file, 258
where clause, 171, 231, 400, 511, 516
Where(), 87, 100
Windows Communication Foundation (WCF), 312–313, 323, 329, 345, 354, 503
Workflow Foundation (WF), 4

X

XElement class, 458
XML
models and, 2
treating a scalar property of type string as XML data, 457
XmlReaders, 257

Entity Framework 4.0 Recipes: A Problem-Solution Approach. Entity Framework 4.0 Recipes: A Problem-Solution Approach, ISBN: 978143022703
Prepared for mallen@iqsolutions.com, Matt Allen
Copyright © 2010 by Larry Tenny and Zeeshan Hirani. This download file is made available for personal use only and is subject to the Terms of Service. Any other use requires prior written consent from the copyright owner. Unauthorized use, reproduction and/or distribution are strictly prohibited and violate applicable laws. All rights reserved.