Oleg Sych

ASP.NET Dynamic Data UNLEASHED

SAMS | 800 East 96th Street, Indianapolis, Indiana 46240 USA

ASP.NET Dynamic Data Unleashed

Copyright © 2012 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-6723-3565-5

ISBN-10: 0-672-33565-4

Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States of America

First Printing: May 2012

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Pearson Education, Inc. cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an "as is" basis. The author and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Bulk Sales

Pearson offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact:

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact:

International Sales
+1-317-581-3793
international@pearsontechgroup.com

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Kristy Hart

Project Editor
Anne Goebel

Copy Editor
Christal White, Language Logistics, LLC

Indexer
Christine Karpeles

Proofreader
Sarah Kearns

Technical Editors
Scott Hunter
David Ebbo

Publishing Coordinator
Cindy Teeters

Cover Designer
Gary Adair

Senior Compositor
Gloria Schurick

Contents at a Glance

Preface xv

PART I Dynamic Data Fundamentals

1 Getting Started 3
2 Entity Framework 37
3 Field Templates 85
4 Entity Templates 117
5 Filter Templates 137
6 Page Templates 153
7 Metadata API 189

PART II Dynamic Data in Real-World Applications

8 Implementing Entity Validation 217
9 Implementing Business Logic 263
10 Building Custom Forms 309
11 Building Dynamic Forms 349
12 Building Custom Search Pages 395
13 Building Dynamic List Pages 421
14 Implementing Security 455

Table of Contents

Preface xv

PART I Dynamic Data Fundamentals

1 Getting Started 3

Creating a New Dynamic Data Application 4
Creating a Data Model 6
Metadata in Entity Framework and Dynamic Data 12
Field Templates and Dynamic Controls 13
Data Annotations 17
Entity Templates 18
Grid Columns and Dynamic Fields 22
Filter Templates 25
Page Templates and URL Routing 29
Summary 35

2 Entity Framework 37

Understanding Entity Model 38
Creating Entity Model 38
ObjectContext and Simple Queries 43
Associations 43
Querying Across Associations 46
Modifying and Saving Entities 46
LINQ to Entities 47
Query Execution 47
Object Identity 48
Lazy and Eager Loading 49
Projections 49
Sorting 50
Grouping 51
Joins 51
Composing Queries 52
SQL Translation 53
Compiled Queries 55
Extension Methods 57

Entity Lifecycle....60
Tracking Changes....60
Optimistic Concurrency Control....61
Transactions....64
Read-Only ObjectContext....65
Entity Model in Depth....66
Mapping Entity Type to Multiple Tables....71
Entity Type Inheritance....72
Inheritance Mapping....73
Stored Procedures....74
Insert, Update, and Delete Stored Procedures....78
Summary....82

3 Field Templates 85

Read-Only Templates....86
Field Value Display Format....87
Edit and Insert Mode Templates....89
Field Value Edit Format....93
Changing Existing Field Template....93
Data Binding....96
Creating a New Field Template....99
Field Template Lookup Rules....104
Data Annotations....104
Type Names....104
Type Aliases....105
Type Fallback....105
Control Mode....106
Mode Fallback....106
Navigation Columns....107
Field Template Lookup Algorithm....108
Built-in Field Templates....111
Boolean....111
Boolean_Edit....111
Children....111
Children_Insert....111
DateTime....112
DateTime_Edit....112
Decimal_Edit....112
EmailAddress....112
Enumeration....112
Enumeration Edit....113
ForeignKey....114

ForeignKey_Edit 114
Integer_Edit 114
ManyToMany 114
ManyToMany_Edit 115
MultilineText_Edit 115
Text 115
Text_Edit 116
Url 116
Summary 116

4 Entity Templates 117

Creating an Entity Template 120
Composite Entity Templates 122
Generic Entity Templates 123
Dynamic Entity Templates 124
Edit and Insert Entity Templates 129
Entity Template Lookup Rules 131
Template Name 131
Template Mode 132
Entity Template Lookup Algorithm 133
Built-in Entity Templates 134
Summary 135

5 Filter Templates 137

Filtering with LINQ 138
Creating a Filter Template 140
GetQueryable Method 143
OnFilterChanged Method 143
DefaultValue Property 143
Dynamic LINQ Queries 143
Building Search Pages with Filter Templates 145
QueryableFilterRepeater Control 148
Filter Template Lookup Rules 150
Built-in Filter Templates 151
Boolean 151
Enumeration 151
ForeignKey 151
Summary 151

6 Page Templates 153

Built-in Page Templates....154
Details Page Template....155
Edit Page Template....159
List Page Template....163
Insert Page Template....169
Creating a New Page Template....172
Custom Page Templates....174
URL Routing....180
Action Route Parameter....180
Table Route Parameter....180
URL Parameters Versus Query Parameters....181
Dynamic Hyperlinks....182
Route Evaluation and Constraints....183
ViewName Route Parameter....185
Page Template Lookup Rules....185
Summary....186

7 Metadata API 189

Metadata Classes at a Glance....189
Metadata Explorer....191
The MetaModel Class....192
Data Model Registration....192
Accessing MetaTable Objects....193
Global Model Registration....195
Scaffolding Support....196
The MetaTable Class....196
Entity Definition....196
Data Annotations....198
Data Access Support....201
Presentation Logic Support....202
Security Support....203
The MetaColumn Class....203
Property Definition....204
Property Kind....206
Data Annotations....207
The MetaForeignKeyColumn Class....211
Presentation Logic....211
The MetaChildrenColumn Class....213
Summary....214

PART II Dynamic Data in Real-World Applications

8 Implementing Entity Validation 217

Validation Framework 217
Validation Error Messages....219
Validation in Entity Framework....221
Customizing Entity Framework Code Generation....227
Replacing Built-in Code Generator with Text Template....227
Customizing Text Template to Change Context Base Class....228
Customizing Text Template to Generate Static Constructors 229
Automated Testing....230
Creating a Unit Test....231
Test Isolation....234
Testing Validation of Different Properties....235
Integration Testing Versus Unit Testing....237
Declarative Validation....238
RequiredAttribute 238
RangeAttribute....239
StringLengthAttribute....241
RegularExpressionAttribute 242
EnumDataTypeAttribute....244
DataTypeAttribute 245
Imperative Validation 245
CustomValidationAttribute 245
Validating Multiple Rules 250
Reporting Multiple Errors....252
Validating Entity State Changes....256
Accessing Original Property Values with ObjectStateManager....256
Accessing Original Property Values in Validation Methods 257
Handling DBNull Values 259
Summary 261

9 Implementing Business Logic 263

Entity Design....264
Measure Impact of Inheritance on Query Performance 264
Keep the Conceptual and Storage Models as Close as Possible....268
Default Property Values 269
Property Interaction Rules 272
OnPropertyChanging Partial Methods....273
ISavableObject Interface 273
Entity Interaction Rules....275
Implementing Entity Interaction Rules in BeforeSave Methods 276

Implementing Entity Interaction Rules as Entity Methods.....277
Implementing Entity Interaction Rules as Context Methods 278
Validation Rules 280
Validation Rule Design 281
Validation of Related Entities.....287
Entity Deletion Rules.....289
Reusing Entity Validation in Business Rule Methods.....290
Saving Changes.....292
Managing Transactions.....293
Invoking Business Rules in Web Applications 294
Error Handling.....296
Business Rules in the Database.....298
Integrating Database Validation in Dynamic Data Applications 299
Implementing Validation Rules in Triggers 304
Implementing Business Rules in Stored Procedures 306
Summary 308

10 Building Custom Forms 309

Multimode Entity Templates 310
Custom Templates 310
Dynamic Templates 312
Creating Dynamic Label Control.....315
Extending DynamicControl 319
Configuring Field Templates.....321
Specifying a Field Template Explicitly 321
Configuring Field Templates Declaratively.....322
Configuring Field Templates Programmatically 323
Creating Custom Field Templates.....324
Implementing Interaction Between Field Templates 327
Accessing Initial Field Values of Other Columns.....330
Accessing Modified Field Values of Other Columns.....331
Initial Implementation of the Region_Edit Field Template.....332
Improving Encapsulation of Field Templates.....334
FieldValue Property.....336
FieldValueChanged Event.....337
AutoPostBack Property 337
Extending Existing Field Templates 337
Using Entity Templates to Control Field Templates 341
Extending DynamicControl (Again) 343
Taking Advantage of the UnleashedControl in Entity Templates....345
Summary 347

11 Building Dynamic Forms 349

Basic Form Layout....350
Configuring Appearance of Labels....352
Prompt Annotation....354
Description Annotation....356
Configuring Field Templates....357
Making Properties Read-Only....357
Overriding Default Field Templates....359
Configuring Appearance of Values....360
Enum Properties and Enumeration Templates....361
Extending the Enumeration Field Templates....362
Custom Data Types and UI Hints....368
Specifying Field Template Parameters....370
Extending Dynamic Data to Support Control Parameters....372
Field Template Interaction in Dynamic Forms....378
Fixing the FindOtherFieldTemplate Method....381
Creating Additional Dynamic Entity Templates....383
Extending Metadata to Support Entity UI Hints....384
Extending Page Templates to Support EntityUIHintAttribute....386
UnleashedEntity Control....386
Building a Dynamic Two-Column Entity Template....388
Summary....393

12 Building Custom Search Pages 395

Displaying Information from Related Entities....396
Column Sorting Across Related Entities....400
Building Custom Search Pages....401
Overcoming Limitations of Dynamic Filters....403
Using Replacement Filter Controls....405
Filtering Based on Related Entities....407
Initializing Nested Filter Control....409
Implementing Query Logic....411
Using the Parent Filter Template....419
Summary....420

13 Building Dynamic List Pages 421

Extending Filter Lookup Rules....422
Extending the FilterFactory Class....423
Extending the MetaTable Class....427
Extending the MetaModel Class....429

Building Dynamic UI with Custom Properties 432
Implementing Support for Custom Properties....433
Model Providers....434
Sorting of Custom Properties....446
Specifying Sort Expression for Custom Properties in Entity Model....446
Extending Dynamic Data to Support Sort Expression Metadata 447
Generating Metadata Attributes 450
Summary 452

14 Implementing Security 455

Table Scaffolding....456
Route Constraints 457
Web.Config Authorization Rules....458
Limitations of Configuration-Based Rules 461
Metadata Security API....462
Modifying Dynamic Templates to Support the Security API....464
Implementing Row-Level Security....472
Security Annotations 476
AuthorizationAttribute 477
CustomQueryAttribute....487
Summary 493

Index 495

About the Author

Oleg Sych has been working as a professional software engineer since 1993, building distributed enterprise applications since 2000, and leading development teams since 2002. Between 2009 and 2011, Oleg received several MVP awards from Microsoft in C# and Visual Studio ALM categories. His professional interests include agile software architecture and development process. In his spare time, Oleg is trying to learn how to ski, play tennis, and raise teenage boys.

Dedication

To Tanyusha, Gleb, and Nickita. I am nothing without you.

Acknowledgments

This book would not be possible without help from Brook Farling, Scott Hunter, David Ebbo, Neil Rowe, and Sven Aelterman. Thank you!

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@samspublishing.com

Mail: Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at informit.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Preface

Most of the web applications that exist today work with data. From a simple guest book page on a personal website, to a large product catalog of an online retailer, to customer relationship management (CRM) and enterprise resource planning (ERP) systems—applications help people work with data. Working with data means dealing with *CRUD*. Every database application must allow users to Create, Read (or Retrieve), Update, and Delete records. A product catalog will not be functional unless users can add new products to the catalog, search for and retrieve existing products, update product information, and delete discontinued products from the catalog. Of course, a product catalog rarely exists by itself; it is usually a part of a larger e-commerce system, which has to support all CRUD operations not only for products, but also for customers, suppliers, shipping companies, orders, and more.

Implementing each type of CRUD operation requires some form of *presentation*, *business*, and *data access logic*. Implementing them in your applications usually requires a lot of repetition. Data access logic for products is similar to data access logic for customers, as well as suppliers and shipping companies. You write SQL or .NET code (or both) to insert, select, update, and delete database records. Presentation logic for products is also similar to presentation logic for customers, suppliers, and shipping companies. You create ASP.NET pages with text boxes, drop-down lists, validation controls, links, and buttons. What truly distinguishes the different types of entities is their purpose—the business logic they represent. The *Product* entity has attributes such as description, price, picture, and rules that define how to validate product names, what to do when a new product is added to the catalog, or when an existing product is discontinued. The *Customer* entity has different attributes—first name, last name, credit card numbers—and rules that define what to do when a customer changes his billing address or requests to close his account.

Understanding the business logic, a developer can usually figure out how to implement the required data access and presentation logic. The resulting application might not look very pretty, but it will perform its business function. The ASP.NET framework provided all building blocks required for implementing database applications, and although it made web developers a lot more productive, it offered little assistance with the repetitive coding of CRUD operations and made it easy for them to fall into the trap of code reuse by copy and paste. It is unfortunate that a typical ASP.NET application has numerous web pages full of validators and code-behind files where data access code is intermixed with business logic.

There are many techniques for reducing code duplication. In ASP.NET, user controls are an excellent way to extract common presentation logic and reuse it in multiple pages. Patterns, such as Domain Model and Active Record (Fowler, 2002), can be used to encapsulate business and data access logic in application code. Some developers also choose to use stored procedures and implement this logic in database code instead. Separation of the business and data access code from the presentation code allows you to reuse it in

multiple web pages without duplication. However, implementing business and data access layers requires more forethought, expertise, and commitment from developers. To be effective, these techniques must be introduced in the application architecture early. Retrofitting them to an existing application requires a lot of refactoring (Fowler, et al., 1999) and can be very costly. It is not unusual for such a refactoring effort to be longer than the initial application implementation itself.

For .NET developers, introduction of LINQ in version 3.5 was a major leap forward. LINQ to SQL and Entity Framework combined the simplicity of *Language-Integrated Queries* with the sophistication of *Metadata Mapping* (Fowler, 2002), making the Domain Model pattern available for mainstream .NET projects. In ASP.NET, both Dynamic Data and MVC can take advantage of the metadata exposed by the LINQ-based frameworks and generate controls or entire pages in ASP.NET applications based on the domain model. Together, these frameworks give web developers powerful, robust, and easy-to-use tools and guidance for implementing data access, business, and presentation logic while reducing code duplication and improving code reuse.

ASP.NET WebForms (with Dynamic Data extensions) and MVC are complementary frameworks that offer unique advantages for different types of web applications. Dynamic Data is a great fit for applications with a large number of similar data-oriented screens, while MVC works better for a small number of unique screens. Most real-world web application projects can benefit from both. Early in the project, Dynamic Data gives you a working user interface, which you can use to discuss the domain model with the business users. As you contine to evolve our understanding of the domain and change the model, Dynamic Data will automatically regenerate the user interface at run time, without forcing you to create and maintain custom web pages. When the domain model is more stable, you can decide which pages have unique requirements and have to be customized. In the end, you may end up replacing some of the web pages with MVC views built on top of the same underlying business and data access logic. The resulting web application can be a mix of a large number of web pages generated by Dynamic Data for maintaining reference data and a small number of unique, public-facing pages implemented in MVC.

The initial version of ASP.NET Dynamic Data, released as part of the .NET Framework 3.5 Service Pack 1, suffered from some short-term limitations and a lingering problem of perception. The fully-featured version, released with the .NET Framework 4, received very little publicity from Microsoft and the developer community focused its attention on the MVC framework instead. This is unfortunate because according to Microsoft's research at the time of this writing, 80% of existing ASP.NET applications are built with WebForms and could take advantage of the productivity improvements Dynamic Data extensions offer without requiring a rewrite in MVC.

This book is meant to correct some of the perceptions, fill the information gap, and help you see Dynamic Data for what it is—a useful tool in the ASP.NET developer's toolbox. The first part offers an in-depth look at the Dynamic Data building blocks, shows how they can be used individually to make routine WebForms programming tasks easier, and

explains how the components fit together to make dynamic generation of entire pages and websites possible. The second part explains how to solve real-world problems—implementing business logic while keeping it separate from the presentation, extending Dynamic Data components and metadata to make them even more powerful, and much more. I hope you find it useful.

Oleg Sych
February 2012
Oldsmar, Florida

This page intentionally left blank

CHAPTER 3

Field Templates

IN THIS CHAPTER

- Read-Only Templates
- Field Value Display Format
- Edit and Insert Mode Templates
- Field Value Edit Format
- Changing Existing Field Template
- Data Binding
- Creating a New Field Template
- Field Template Lookup Rules
- Built-in Field Templates

Enterprise applications deal with employees, managers, emergency contacts, prospects, customers, suppliers, and vendors—in other words, people and organizations, who all have names, addresses, and phone numbers. Implementing data entry logic for these attributes is repetitive, and for years, the ASP.NET developers have been developing user controls to reduce the code duplication. It is common to see user controls with text boxes and validators for entering street, city, state, and postal code in many web applications today.

Although they are a great mechanism for reusing presentation and validation logic, user controls require a certain amount of plumbing before they can be used on a web page. At the very minimum, user controls must be registered, and because even the simplest controls need to provide slightly varying functionality in different contexts, they also usually have one or more properties that need to be configured. As the user controls get more and more granular, the amount of plumbing work required to apply them increases. It quickly reaches the point of diminishing returns, and developers usually stop short of implementing user controls for the individual data entry fields.

Field template is a special type of ASP.NET user control that encapsulates presentation and validation logic for a single field. Unlike the traditional user controls, field templates do not require registration. Instead, a simple `DynamicControl` or a `DynamicField` placed on a web page automatically loads and configures an appropriate field template based the column name and metadata.

Metadata information, readily available in Dynamic Data web applications, dramatically lowers the amount of plumbing code developers have to write when working with user controls, enables effective use of smaller user controls, and takes code reuse to an unprecedented new level.

Read-Only Templates

By convention, field templates are located in the `DynamicData\FieldTemplates` folder of the Dynamic Data web application projects and websites. A single instance of a field template provides user interface for one particular field value. Consider Listing 3.1, which shows markup file of the `DateTime` field template.

LISTING 3.1 DateTime Field Template (Markup)

```
<%@ Control Language="C#" CodeBehind="DateTime.ascx.cs"
  Inherits="WebApplication.DynamicData.FieldTemplates.DateTimeField" %>

<asp:Literal runat="server" ID="literal" Text="<%# FieldValueString %>" />
```

This is a *read-only* field template, the simplest type of field template. It relies on ASP.NET data binding syntax `<%# %>` to display the field value in a single `Literal` control. Listing 3.2 shows the code-behind of the `DateTime` field template. You can see that it inherits from the `FieldTemplateUserControl`, a special base class provided for field templates by Dynamic Data. This class defines the `FieldValueString` property to which the `Literal` control is bound.

LISTING 3.2 DateTime Field Template (Code-behind)

```
using System.Web.DynamicData;
using System.Web.UI;

namespace WebApplication.DynamicData.FieldTemplates
{
  public partial class DateTimeField : FieldTemplateUserControl
  {
    public override Control DataControl
    {
      get { return this.literal; }
    }
  }
}
```

Code-behind files perform most of the work required to make the generated web pages dynamic. In this simplest example, the template overrides the `DataControl` property defined by the base class. Entity templates (see Chapter 4, "Entity Templates") use this property when dynamically generating entity forms, to associate each field template with a matching `Label` control.

Field Value Display Format

In addition to the `FieldValueString` property, the `FieldTemplateUserControl` class also provides another property called `FieldValue`, which returns the raw value object. At a cursory glance, the `FieldValue` property could be used to bind the `Literal` control as well. However, it is important to use the `FieldValueString` property as it automatically takes into account additional display format properties when converting the raw field value to a string:

- `DataFormatString` specifies a .NET format string that will be passed to the `String.Format` method to convert the field value to a display string.
- `NullDisplayText` specifies a replacement string that will be used when the raw field value is null.
- `ConvertEmptyStringToNull` is a Boolean value that determines whether an empty string value should treated as null and if the `NullDisplayText` will be used as a replacement for empty strings as well as null values. This property is `true` by default.
- `HtmlEncode` is a Boolean value that determines whether any special HTML symbols in the raw field value, such as `'<'` or `'>'`, will be encoded using escape sequences, such as `<` and `>`. This property is `true` by default.

Display format properties can be specified in the data model by applying the `DataTypeAttribute` or the `DisplayFormatAttribute` to a particular column, as shown in Listing 3.3. The `DisplayFormatAttribute`, applied to the OrderDate column, specifies a long date `{0:D}` format string. The `DataTypeAttribute`, applied to the RequiredDate column, does not have its own `DataFormatString` property. Instead, it offers the `DisplayFormat` property, which gets a default `DisplayFormatAttribute` generated, based on the specified data type. For `DataType.Date`, the default data format string will be a short date `{0:d}`.

LISTING 3.3 Specifying Display Format Using Data Annotations

```
using System.ComponentModel.DataAnnotations;

namespace DataModel
{
  [MetadataType (typeof(Order.Metadata))]
  partial class Order
  {
```

LISTING 3.3 Continued

```
    public class Metadata
    {
      [DisplayFormat(DataFormatString = "{0:D}")]
      public object OrderDate;

      [DataType(DataType.Date)]
      public object RequiredDate;
    }
  }
}
```

Display format properties can be also specified in page markup by setting the corresponding properties of the `DynamicControl` or `DynamicField` classes, similar to the properties of the "non-dynamic" `BoundField`. Listing 3.4 shows an example where a `DynamicField` is used in a `GridView` control and specifies a custom date format string `{0:MM/dd/yy}` for the ShippedDate column.

LISTING 3.4 Specifying Data Format Using DynamicField

```
<%@ Page Language="C#"
  MasterPageFile="~/Site.master" CodeBehind="SamplePage.aspx.cs"
  Inherits="WebApplication.Samples.Ch01.DataFormatString.SamplePage" %>

<asp:Content ContentPlaceHolderID="ContentPlaceHolder1" runat="server">
  <asp:GridView ID="gridView" runat="server" DataSourceID="dataSource"
    AutoGenerateColumns="false" AllowPaging="true">
    <columns>
      <asp:DynamicField DataField="OrderDate" />
      <asp:DynamicField DataField="RequiredDate" />
      <asp:DynamicField DataField="ShippedDate" DataFormatString="{0:MM/dd/yy}" />
    </columns>
  </asp:GridView>
  <asp:EntityDataSource ID="dataSource" runat="server"
    ConnectionString="name=NorthwindEntities"
    DefaultContainerName="NorthwindEntities" EntitySetName="Orders" />
</asp:Content>
```

Figure 3.1 shows the web page generated by this code. As you can see, the `OrderDate` values use the long format specified in the `DisplayFormatAttribute` in the model; the `RequiredDate` values use the short format provided by the `DataTypeAttribute`; and the `ShippedDate` values use the custom format specified in the page markup.

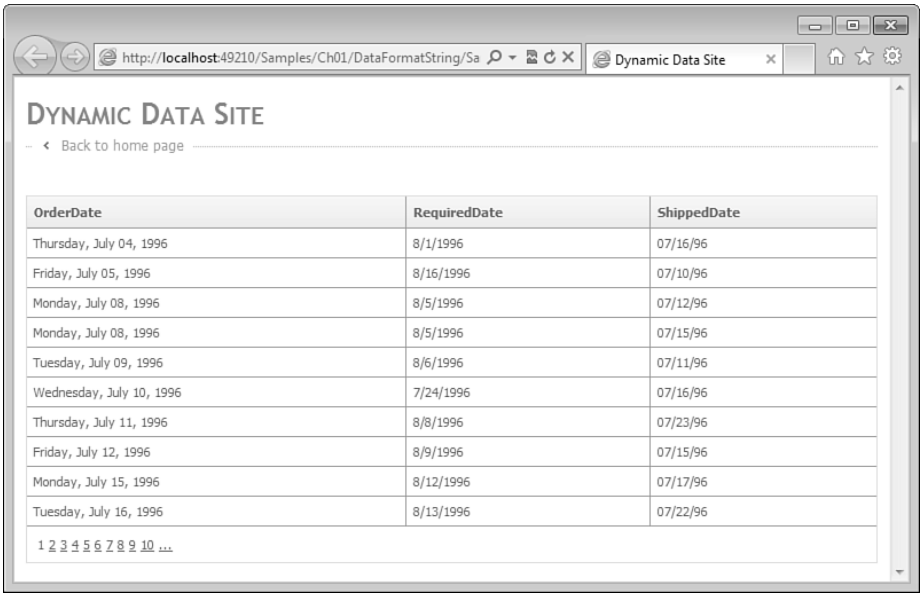


FIGURE 3.1 GridView with `DataType` and `DisplayFormat` attributes applied.

Inside of the field templates, the display format properties are available via the `FormattingOptions` property inherited from the `FieldTemplateUserControl`. The base class automatically initializes them based on the formatting options of the host—`DynamicControl` or `DynamicField`.

As you might expect, a fair amount of logic is involved in converting the raw field value to the final display string. By using the `FieldValueString` property of the `FieldTemplateUserControl` class, field templates ensure consistent behavior that offers application developers great flexibility in defining the presentation aspects of their data. The `DataTypeAttribute` allows specifying default display properties for a column in the data model based on its data type. The `DisplayFormatAttribute` allows fine-tuning it with the standard or custom format strings. Finally, the `DynamicControl` and `DynamicField` controls allow tailoring the display of column values for a specific page in a web application.

Edit and Insert Mode Templates

In addition to the read-only field templates, Dynamic Data has *Edit* and *Insert* field templates. Unlike the Read-Only templates, where the field value is simply displayed on the page, the Edit templates allow the user to modify it using a data entry control, such as a `TextBox`. By convention, Dynamic Data distinguishes Edit and Insert templates by the `_Edit` and `_Insert` they have at the end of their file names, respectively. The Insert templates are a special case of Edit templates, used for editing field values of rows that have not been saved to the database. However, most Edit mode field templates can be used in Insert mode as well.

The Edit field templates are significantly more complex than their read-only counterparts. Consider Listing 3.5, which shows the Edit version of the `DateTime` field template.

LISTING 3.5 DateTime_Edit Field Template (Markup)

```
<%@ Control Language="C#" CodeBehind="DateTime_Edit.ascx.cs"
  Inherits="WebApplication.DynamicData.FieldTemplates.DateTime_EditField" %>

<asp:TextBox ID="textBox" runat="server" Text='<%# FieldValueEditString %>'
  Columns="20" />

<asp:RequiredFieldValidator runat="server" ID="requiredFieldValidator"
  ControlToValidate="textBox" Enabled="false" />
<asp:RegularExpressionValidator runat="server" ID="regularExpressionValidator"
  ControlToValidate="textBox" Enabled="false" />
<asp:DynamicValidator runat="server" ID="dynamicValidator"
  ControlToValidate="textBox" />
<asp:CustomValidator runat="server" ID="dateValidator" ControlToValidate="textBox"
  EnableClientScript="false" Enabled="false"
  OnServerValidate="DateValidator_ServerValidate" />
```

In addition to the data entry control, the edit field templates typically have one or more validation controls. The `DateTime_Edit` template includes a `RequiredFieldValidator`, a `RegularExpressionValidator`, a `DynamicValidator`, and a `CustomValidator` control. Notice that with the exception of the `DynamicValidator`, these controls are not enabled by default. They are configured dynamically by the `Page_Load` event handler defined in the code-behind file shown in Listing 3.6.

Each validator control is configured by a call to the `SetUpValidator` method of the `FieldTemplateUserControl` base class. This method enables and configures the validator if the column metadata contains an appropriate data annotation attribute. In particular, the `RequiredFieldValidator` will be enabled when the column is marked with a `RequiredAttribute` that does not allow empty strings, and the `RegularExpressionValidator` will be enabled when the column is marked with a `RegularExpressionAttribute`. In addition to enabling the validator, the `SetUpValidator` method also assigns its `ErrorMessage` and `ToolTip` properties with an error message either explicitly specified in the data annotation attribute or generated based on the column's display name.

`DynamicValidator` is a special validation control provided by Dynamic Data. It ensures all validation attributes applied to the column are invoked to verify its value is correct. This andincludes not only the well-known `RequiredAttribute`, `RangeAttribute`, and `CustomValidationAttribute`, but also all other descendants of the `ValidationAttribute` that may be applied to the column and are not covered by a specific ASP.NET validator control in this field template.

The CustomValidator control in this field template performs the job you would normally expect the DataTypeAttribute to do. Even though DataTypeAttribute inherits from the ValidationAttribute class, it does not actually perform any validation. Its IsValid method always returns true, which is why the DateTime_Edit template uses the CustomValidator control to ensure that the value entered by the user matches the type specified by the DataTypeAttribute. The SetUpCustomValidator method enables the validator if the column was marked with a DataTypeAttribute and the DateValidator_ServerValidate method performs the actual validation.

LISTING 3.6 DateTime_Edit Field Template (Code-Behind)

```
using System;
using System.Collections.Specialized;
using System.ComponentModel.DataAnnotations;
using System.Web;
using System.Web.DynamicData;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WebApplication.DynamicData.FieldTemplates
{
  public partial class DateTime_EditField : FieldTemplateUserControl
  {
    private static DataTypeAttribute DefaultDateAttribute =
      new DataTypeAttribute(DataType.DateTime);

    public override Control DataControl
    {
      get { return this.textBox; }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
      this.textBox.ToolTip = this.Column.Description;

      this.SetUpValidator(this.requiredFieldValidator);
      this.SetUpValidator(this.regularExpressionValidator);
      this.SetUpValidator(this.dynamicValidator);
      this.SetUpCustomValidator(this.dateValidator);
    }

    protected override void ExtractValues(IOrderedDictionary dictionary)
    {
      dictionary[Column.Name] = this.ConvertEditedValue(this.textBox.Text);
    }
```

LISTING 3.6 Continued

```
    protected void DateValidator_ServerValidate(object source,
      ServerValidateEventArgs args)
    {
      DateTime dummyResult;
      args.IsValid = DateTime.TryParse(args.Value, out dummyResult);
    }

    private void SetUpCustomValidator(CustomValidator validator)
    {
      if (this.Column.DataTypeAttribute != null)
      {
        switch (this.Column.DataTypeAttribute.DataType)
        {
          case DataType.Date:
          case DataType.DateTime:
          case DataType.Time:
            this.EnableCustomValidator(validator, this.Column.DataTypeAttribute);
            break;
        }
      }
      else if (this.Column.ColumnType.Equals(typeof(DateTime)))
      {
        this.EnableCustomValidator(validator, DefaultDateAttribute);
      }
    }

    private void EnableCustomValidator(CustomValidator validator,
      DataTypeAttribute attribute)
    {
      validator.Enabled = true;
      validator.ErrorMessage = HttpUtility.HtmlEncode(
        attribute.FormatErrorMessage(this.Column.DisplayName));
    }
  }
}
```

Field Value Edit Format

You might have noticed that the `DateTime_Edit` template uses the `FieldValueEditString` property of the `FieldTemplateUserControl` base class. This property is similar to the `FieldValueString` property discussed earlier in that it respects the `FormattingOptions` when converting the raw column value to a display string. However, it was specifically designed for use in the Edit mode templates, and an additional property is used to control its behavior.

- `ApplyFormatInEditMode` is a Boolean value that determines whether the `DataFormatString`, `NullDisplayText`, `ConvertEmptyStringToNull`, and `HtmlEncode` display format properties will be used in Edit mode. This property is `false` by default.

Similar to using the `FieldValueString` property when implementing Read-only field templates, it is also important to rely on the `FieldValueEditString` property as much as possible when implementing the Edit and Insert mode templates. This helps the field templates to behave consistently and allow the application developers to change presentation aspects of column values using metadata attributes as well as explicit configuration in page markup.

Changing Existing Field Template

Dynamic Data uses the same field template for all matching columns in the model. The `DateTime_Edit` field template is automatically used for all columns of type `DateTime` in the Northwind data model, including OrderDate, RequiredDate, and ShippedDate columns of the Order table as well as BirthDate and HireDate columns of the Employee table. By changing definition of the field template, you affect all web pages where it is used. Figure 3.2 shows how the experience of entering date values can be improved by allowing the users to select a date from a calendar drop-down.

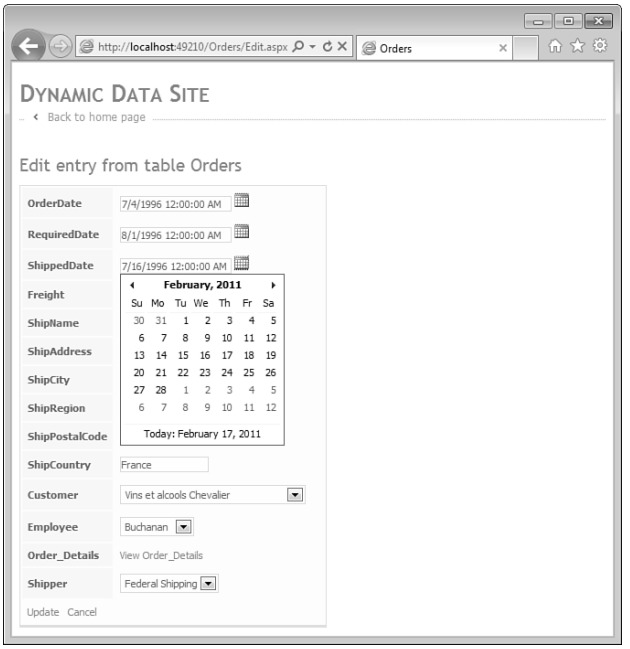


FIGURE 3.2 Modified DateTime field template with calendar drop-down.

The modified field template, `DateTime_Edit.ascx`, is shown in Listing 3.7. This example uses the `CalendarExtender` control from the AJAX Control Toolkit, but you can use any other ASP.NET control from your favorite component vendor. The `CalendarExtender` is associated with the `TextBox` by specifying its `TagetControlID` property. Using the `PopupButtonID` property, it is also associated with an `ImageButton` that users can click to display the calendar drop-down.

NOTE

The AJAX (Asynchronous JavaScript and XML) Control Toolkit is an extension of the ASP.NET framework developed by Microsoft to enhance client-side behavior of the standard WebForms controls. It is an open source project hosted on CodePlex and available for download at `http://AjaxControlToolkit.CodePlex.com`.

The AJAX Control Toolkit comes in a ZIP archive that you need to extract to a directory on your hard drive. Before you can begin using AJAX controls, you need to add the downloaded AjaxControlToolkit.dll as a reference to your web application project in Visual Studio.

LISTING 3.7 Extended DateTime_Edit Field Template (Markup)

```
<%@ Control Language="C#" CodeBehind="DateTime_Edit.ascx.cs"
  Inherits="WebApplication.DynamicData.FieldTemplates.DateTime_EditField" %>

<asp:TextBox ID="textBox" runat="server" Text='<%# FieldValueEditString %>'
  Columns="20" />
<asp:ImageButton ID="calendarButton" runat="server"
  ImageUrl="../Content/Images/Calendar.png" />
<ajax:CalendarExtender runat="server" TargetControlID="textBox"
  PopupButtonID="calendarButton" />

<asp:RequiredFieldValidator runat="server" ID="requiredFieldValidator"
  ControlToValidate="textBox" Enabled="false" />
<asp:RegularExpressionValidator runat="server" ID="regularExpressionValidator"
  ControlToValidate="textBox" Enabled="false" />
<asp:DynamicValidator runat="server" ID="dynamicValidator"
  ControlToValidate="textBox" />
<asp:CustomValidator runat="server" ID="dateValidator" ControlToValidate="textBox"
  EnableClientScript="false" Enabled="false"
  OnServerValidate="DateValidator_ServerValidate" />
```

Notice that this example uses a custom `ajax:` tag prefix with the `CalendarExtender` without actually registering it. We could have used the ASP.NET `<%@ Register %>` directive to register this tag prefix directly in the `DateTime_Edit.ascx`. However, because the AJAX Control Toolkit is used in other field templates of this sample project, it is better to register it in the `Web.config` as shown in Listing 3.8. This makes the AJAX controls available in all pages and controls of the web application and helps you avoid having to register the same tag prefix in multiple pages and user controls.

LISTING 3.8 Registering AJAX Controls in the Web Configuration File

```
<?xml version="1.0"?>
<configuration>
  <system.web>
    <compilation debug="true" targetFramework="4.0"/>
    <pages styleSheetTheme="Default">
      <controls>
        <add tagPrefix="ajax" assembly="AjaxControlToolkit"
          namespace="AjaxControlToolkit"/>
      </controls>
    </pages>
  </system.web>
  <system.webServer>
    <modules runAllManagedModulesForAllRequests="true"/>
  </system.webServer>
  <connectionStrings>
```

LISTING 3.8 Continued

```
    <add name="NorthwindEntities"
      providerName="System.Data.EntityClient"
      connectionString="
        metadata=res://*/Northwind.csdl¦
                 res://*/Northwind.ssdl¦
                 res://*/Northwind.msl;
        provider=System.Data.SqlClient;
        provider connection string=&quot;
          data source=.\sqlexpress;
          initial catalog=Northwind;
          integrated security=True;
          multipleactiveresultsets=True&quot;"/>
  </connectionStrings>
</configuration>
```

Data Binding

Several of the built-in Dynamic Data field templates, including the `DateTime` and `DateTime_Edit` templates, rely on data binding to display column value on the page.

```
<asp:TextBox ID="textBox" runat="server" Text='<%# FieldValueEditString %>' />
```

In addition, the field template controls themselves are data bound, and several properties of the `FieldTemplateUserControl` base class rely on the data binding mechanics. These properties can be used safely *only* from the `DataBind` method and will throw an `InvalidOperationException` when used outside of the data binding context, such as during a post-back:

- `Row` provides access to the entity instance the field template is bound to, such as an Order object. In Read-only and Edit modes, it is an equivalent of calling the `Page.GetDataItem` method. In Insert mode, this property returns a `CustomTypeDescriptor` object that provides access to default field values extracted by the page template from the page request URL (more on this in Chapter 6, "Page Templates").
- `FieldValue` returns value of the column property, such as `Order.OrderDate`. It is an equivalent of calling `DataBinder.Eval(Row, Column.Name)`.
- `FieldValueString` converts the field value to a display string in Read-only mode, using the formatting options specified in the data model or page markup. It is an equivalent of calling `FormattingOptions.FormatValue(FieldValue)`.

- `FieldValueEditString` converts the field value to a string in Edit or Insert mode, using the formatting options specified in the data model or page markup. It is an equivalent of calling `FormattingOptions.FormatEditField(FieldValue)`.

During a post-back, a field template itself does not actually save the new column value back in its `Row` object. Instead, the `FieldTemplateUserControl` class implements the `IBindableControl` interface and relies on the parent `FormView`, `DetailsView`, and `GridView` controls to perform this task. When the parent control is handling an Insert or an Update command, it iterates through the list of its child controls and calls the `ExtractValues` of each child that implements `IBindableControl` interface.

```
protected override void ExtractValues(IOrderedDictionary dictionary)
{
  dictionary[this.Column.Name] =
    this.ConvertEditedValue(this.textBox.Text);
}
```

The `ExtractValues` method receives a dictionary object used to store column name and value pairs. The extract from the `DateTime_Edit` field template just shown is a typical implementation of this method. The `ConvertEditedValue` method is provided by the `FieldTemplateUserControl` base class. It can automatically convert the field value to null based on the `ConvertEmptyStringToNull` and `NullDisplayText` formatting options.

Notice that by providing the dictionary argument, `ExtractValue` method allows a field template to return values of other columns or even multiple column values. This capability is important for the foreign key templates used for navigation properties, such as the Customer property of the `Order` class in the Northwind data model, that need to return values of the corresponding primitive properties, such as CustomerID. Consider Listing 3.9, which shows the markup of the `ForeignKey_Edit` template.

LISTING 3.9 ForeignKey_Edit Field Template (Markup)

```
<%@ Control Language="C#" CodeBehind="ForeignKey_Edit.ascx.cs"
  Inherits="WebApplication.DynamicData.FieldTemplates.ForeignKey_EditField" %>

<asp:DropDownList ID="dropDownList" runat="server"/>

<asp:RequiredFieldValidator runat="server" ID="requiredFieldValidator"
  ControlToValidate="dropDownList" Enabled="false" />
<asp:DynamicValidator runat="server" ID="dynamicValidator"
  ControlToValidate="dropDownList" />
```

Notice that the `ForeignKey_Edit` template uses a `DropDownList` control, which allows users to select from a list of items created programmatically, in the `Page_Load` method shown in Listing 3.10. Each `ListItem` object in the `DropDownList` represents a single foreign key, with `Text` property storing value of the display column from the referenced

table and `Value` property storing a comma-separated list of its primary key columns. This code relies on the convenience method, called `PopulateListControl`, provided by the `FieldTemplateUserControl` base class, which does all the heavy lifting required to query the referenced table, create the `ListItem` objects, and add them to the `DropDownList`.

LISTING 3.10 ForeignKey_Edit Field Template (Code-Behind)

```
using System;
using System.Collections.Specialized;
using System.Web.DynamicData;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WebApplication.DynamicData.FieldTemplates
{
  public partial class ForeignKey_EditField : FieldTemplateUserControl
  {
    public override Control DataControl
    {
      get { return this.dropDownList; }
    }

    protected void Page_Load(object sender, EventArgs e)
    {
      if (this.dropDownList.Items.Count == 0)
      {
        if (this.Mode == DataBoundControlMode.Insert ¦¦ !this.Column.IsRequired)
          this.dropDownList.Items.Add(new ListItem("[Not Set]", string.Empty));
        this.PopulateListControl(this.dropDownList);
      }

      this.SetUpValidator(this.requiredFieldValidator);
      this.SetUpValidator(this.dynamicValidator);
    }

    protected override void OnDataBinding(EventArgs e)
    {
      base.OnDataBinding(e);
      string selectedValueString = this.GetSelectedValueString();
      if (this.dropDownList.Items.FindByValue(selectedValueString) != null)
        this.dropDownList.SelectedValue = selectedValueString;
```

```
    }

    protected override void ExtractValues(IOrderedDictionary dictionary)
    {
      this.ExtractForeignKey(dictionary, this.dropDownList.SelectedValue);
    }
  }
}
```

To display the current field value, the `ForeignKey_Edit` template cannot rely on the `FieldValueEditString` property because the `FieldValue` in this case consists of one or more values of the underlying primitive columns. For the Customer column of the Order table, this would mean displaying CustomerID, "ALFKI", instead of the Customer's CompanyName, "Alfreds Futterkiste", which is not very user friendly. Instead, the `ForeignKey_Edit` template overrides the `OnDataBinding` method to select the matching item in the `DropDownList`. It calls the `GetSelectedValueString`, a convenience method provided by the `FieldTemplateUserControl` base class. For foreign key columns, this method is equivalent to calling `ForeignKeyColumn.GetForeignKeyString(Row)` and returns a comma-separated list of values of the primitive columns in the foreign key.

The `ForeignKey_Edit` template also overrides the `ExtractValues` method and calls the `ExtractForeignKey` provided by the `FieldTemplateUserControl` base class. Given a comma-separated list of column values, this method stores them in the dictionary under the names of the foreign key columns. Under the hood, this method simply calls `ForeignKeyColumn.ExtractForeignKey(dictionary, selectedValue)`.

Creating a New Field Template

The set of field templates provided by Dynamic Data can be extended whenever a special data entry control is needed to handle values of a certain type. For example, the Northwind database uses regular string columns to store phone numbers for customers, employees, suppliers, and shippers. The phone numbers already stored in the database are in a consistent `(###)###-####` format; however, Dynamic Data uses the regular `Text_Edit` field template for these columns, which does not enforce the phone number format. It would be nice to create a field template that would help users enter the phone numbers correctly.

You create a new field template by adding a new user control to the `DynamicData\FieldTemplates` folder of your project. You can also copy an existing field template that most closely matches the desired functionality. If you do copy an existing field template, remember to change the class name in both markup and code-behind files. In this example, however, we will start from scratch and create a new user control called `PhoneNumber_Edit`, shown in Listing 3.11.

LISTING 3.11 PhoneNumber_Edit Field Template (Markup)

```
<%@ Control Language="C#" CodeBehind="PhoneNumber_Edit.ascx.cs"
  Inherits="WebApplication.DynamicData.FieldTemplates.PhoneNumberEditField" %>

<asp:TextBox ID="textBox" runat="server" Text='<%# FieldValueEditString %>'/>
<ajax:FilteredTextBoxExtender TargetControlID="textBox" runat="server"
  FilterType="Custom" ValidChars="+()-1234567890" />

<asp:RequiredFieldValidator runat="server" ID="requiredFieldValidator"
  ControlToValidate="textBox" Enabled="false" />
<asp:RegularExpressionValidator runat="server" ID="regularExpressionValidator"
  ControlToValidate="textBox" />
<asp:DynamicValidator runat="server" ID="dynamicValidator"
  ControlToValidate="textBox" />
```

As you can see, the `PhoneNumber_Edit` field template is similar to the `DateTime` template discussed earlier. It has a `TextBox` control used to enter the phone number, a `FilteredTextBoxExtender`, another AJAX control used to prevent users from entering symbols that would not be valid in a phone number, as well as several validator controls.

Although having a single `DynamicValidator` control is sufficient to enforce all validation attributes applied to the column in the data model, this control performs validation only on the server side, which requires a post-back before the error can be reported. Other ASP.NET validation controls, such as the `RequiredFieldValidator`, are optional, but because they also offer client-side validation logic, it's best to include them in field templates to provide immediate feedback to the user, and improve the data entry experience.

Listing 3.12 shows the code-behind of the `PhoneNumber_Edit` field template.

LISTING 3.12 PhoneNumber_Edit Field Template (Code-Behind)

```
using System;
using System.Collections.Specialized;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Web;
using System.Web.DynamicData;
using System.Web.UI;
using System.Web.UI.WebControls;

namespace WebApplication.DynamicData.FieldTemplates
{
  public partial class PhoneNumberEditField : FieldTemplateUserControl
  {
    public override Control DataControl
    {
```

```
    get { return this.textBox; }
  }

  protected void Page_Load(object sender, EventArgs e)
  {
    this.textBox.ToolTip = this.Column.Description;

    this.SetUpValidator(this.requiredFieldValidator);
    this.SetUpValidator(this.dynamicValidator);
    this.SetUpRegexValidator(this.regularExpressionValidator);
  }

  protected override void ExtractValues(IOrderedDictionary dictionary)
  {
    dictionary[this.Column.Name] = this.ConvertEditedValue(this.textBox.Text);
  }

  private void SetUpRegexValidator(RegularExpressionValidator validator)
  {
    RegularExpressionAttribute attribute = this.Column.Attributes
      .OfType<RegularExpressionAttribute>().FirstOrDefault();
    if (attribute == null)
    {
      attribute = new RegularExpressionAttribute(@"\(\d{3}\)\d{3}-\d{4}");
      attribute.ErrorMessage = "The field {0} must be a valid phone number.";
    }

    validator.Enabled = true;
    validator.Text = "*";
    validator.ValidationExpression = attribute.Pattern;
    validator.ErrorMessage = HttpUtility.HtmlEncode(
      attribute.FormatErrorMessage(this.Column.DisplayName));
    validator.ToolTip = validator.ErrorMessage;

    this.IgnoreModelValidationAttribute(attribute.GetType());
  }
 }
}
```

Most of the phone number validation in the new field template is performed by the standard ASP.NET control, `RegularExpressionValidator`, which ensures that the value entered by the user matches the regular expression pattern that defines a valid phone number. In addition to the server-side validation based on the .NET implementation of regular expressions in the `System.Text.RegularExpressions` namespace, it also works on the client side taking advantage of the regular expression functionality in JavaScript.

The RegularExpressionValidator is configured by the SetUpRegexValidator method defined in the code-behind. This method first searches the list of attributes applied to the column in the data model, Column.Attributes, for a RegularExpressionAttribute that might have been supplied by the application developer to indicate the expected phone number pattern. If no such attribute exists, this method creates a default RegularExpressionAttribute with the pattern, \(\d{3}\)\d{3}-\d{4}, that matches U.S. phone numbers, such as (877)514-9180, with a three-digit area code in parentheses, followed by a seven-digit local number. The default attribute also has a generic error message that says that a column must be a valid phone number.

Having obtained the RegularExpressionAttribute instance, the SetUpRegexValidator method configures the RegularExpressionValidator control consistently with how the SetUpValidator method of the FieldTemplateUserControl base class does it—the Text displayed by the validator itself is always a star symbol; its ErrorMessage is reported through a ValidationSummary control on the page and the ToolTip contains the complete error message.

The last step in setting up the RegularExpressionValidator is to call the IgnoreModelValidationAttribute method of the FieldTemplateUserControl base class. This allows the DynamicValidator to ignore the RegularExpressionAttribute during its validation because this attribute is already covered by the RegularExpressionValidator. Aside from helping to avoid doing the same validation check twice, this also prevents the regular expression validation error from being reported twice—first by the RegularExpressionValidator and then by the DynamicValidator control.

Listing 3.13 shows an example of how the new PhoneNumber_Edit field template can be associated with a column using data annotation attributes.

LISTING 3.13 Associating PhoneNumber Template with a Column

```
using System.ComponentModel.DataAnnotations;

namespace DataModel
{
  [MetadataType(typeof(Employee.Metadata))]
  partial class Employee
  {
    public class Metadata
    {
      [DataType(DataType.Date)]
      public object BirthDate;

      [DataType(DataType.Date)]
      public object HireDate;

      [DataType(DataType.PhoneNumber)]
      [RegularExpression(@"\(\d{2,3}\)\d{3}-\d{4}",
        ErrorMessage = "The Home Phone must be a valid phone number.")]
```

```
        public object HomePhone;
    }
  }
}
```

`DataTypeAttribute`, applied to the HomePhone column, specifies this string column's actual data type is `PhoneNumber`. Dynamic Data uses this information as a hint when looking for a matching field template for this column and chooses the new `PhoneNumber_Edit` template instead of the default `Text_Edit`. In this example, `RegularExpressionAttribute` is also applied to the HomePhone column to provide a pattern that matches not only U.S. phone numbers, which have three-digit area codes, but also UK numbers, which might have two-digit area codes.

Figure 3.3 shows the Employee Edit page after the implementation of the `PhoneNumber_Edit` field template has been completed. Although you cannot tell from this screenshot, thanks to the `FilteredTextBoxExtender`, you cannot type any alphabetical symbols in the phone number text box. The pattern validation is performed on the client side as soon as you tab out of the text box, such as in this case, where one digit was removed from an otherwise valid UK phone number.

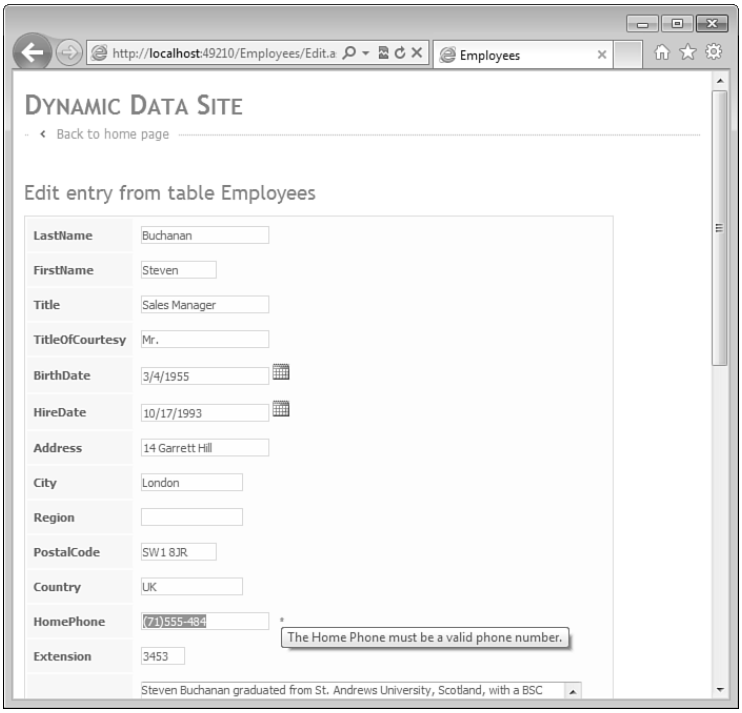


FIGURE 3.3 Employee Edit page with the New PhoneNumber_Edit field template.

Field Template Lookup Rules

Dynamic Data tries to find the most specific template that matches the field based on the column type and control mode. If such a field template does not exist, Dynamic Data will gradually relax the matching rules until it finds the best possible match among the field templates in the project.

Data Annotations

Consider the RequiredDate column of the Order table used in Listing 3.3 earlier in this chapter. If the developer specifies the exact template name by applying the `UIHintAttribute` as shown in Listing 3.14, Dynamic Data first checks to see if a template with the specified name, `MyDate.ascx`, exists. If this template does not exist, Dynamic Data then checks if a `DataTypeAttribute` is applied to the column and tries using the specified `DataType` value as the template name. In this example, it checks to see if a field template called `Date.ascx` exists.

LISTING 3.14 Data Annotations That Affect Field Template Lookup

```
using System.ComponentModel.DataAnnotations;

namespace DataModel
{
  [MetadataType(typeof(Order.Metadata))]
  partial class Order
  {
    public class Metadata
    {
      [UIHint("MyDate")]
      [DataType(DataType.Date)]
      public object RequiredDate;
    }
  }
}
```

Type Names

If the column has no data annotation attributes, Dynamic Data then tries to find a matching template based on the name of its type. First, it looks for a template whose name matches the full name of the type, including the namespace where the type is defined. In the example of `RequiredDate`, when looking for a template for a column of `DateTime` type, it checks to see if a template called `System.DateTime.ascx` exists. If not, Dynamic Data then tries the short type name, without the namespace, and finally chooses the template called `DateTime.ascx` provided by Dynamic Data out of the box.

Type Aliases

Dynamic Data allows template names to use special aliases instead of actual type names for two built-in .NET types, `Int32` and `String`. When looking for a field template for a column of type `String`, Dynamic Data considers the template called `Text.ascx` to be a match. In the Northwind data model, this means that the `Text.ascx` template will be used for the ProductName column of the Product table. Likewise, when looking for a template for an `Int32` column, Dynamic Data considers `Integer.ascx` to be a match as well. Table 3.1 shows the built-in type aliases at a glance.

TABLE 3.1 Type Name Aliases

Type Name	Alias
Int32	Integer
String	Text

Type aliases are considered less specific than the type names, and type *names* take precedence over *aliases* during template lookup. For example, if the Dynamic Data project contains templates called `Int32.ascx` and `Integer.ascx`, only the one called `Int32.ascx` will be used for an integer column.

Type Fallback

If there is no matching field template for a particular data type, Dynamic Data uses *fallback logic* to find a template for a more generic type that could be used instead. Consider the UnitsInStock column of the Product entity, which is of type `Int16`. When looking for a field template for this column, Dynamic Data first checks to see if a template called `Int16.ascx` exists. Because there is no template with this name, it uses the fallback rules shown in Table 3.2 to determine the fallback type, `Int32`, and tries to find a matching template for that type.

TABLE 3.2 Type Fallback Rules

Data Type	Fallback Type
`Double, Float`	`Decimal`
`Byte, Int16, Int64`	`Int32`
`Char, Decimal, DateTime, DateTimeOffset, Guid Int32, TimeSpan`	`String`

Both type names and aliases are used during template lookup for the fallback type, just as they were used for the original type. For the UnitsInStock column, this means that Dynamic Data will then try `System.Int32.ascx`, `Int32.ascx`, and `Integer.ascx`. Because none of these field templates exist in the default Dynamic Data project, it falls back to the next type—`String` and tries `System.String.ascx`, `String.ascx`, and finally choosing `Text.ascx`.

Control Mode

Control mode defines the context where the field template will be used. It can be one of the members of the ASP.NET `DataBoundControlMode` enumeration:

- `ReadOnly` templates are used by the `Details.aspx` page template or custom pages to display field values in Read-only mode.
- `Edit` templates are used by the `Edit.aspx` page template or custom pages to change field values of an existing row.
- `Insert` are used by the `Insert.aspx` page template or custom pages to collect field values for a new row.

When looking for a `ReadOnly` field template for the ShippedDate column of the Order table, which has `DateTime` data type and no additional metadata attributes in the sample data model, Dynamic Data finds the `DateTime.ascx` template. `Edit` mode is considered more specific because it requires the field template to not only display the current field value, but also allow changing it. Therefore, when looking for an Edit field template for the ShippedDate column, Dynamic Data chooses the template called `DateTime_Edit.ascx` if it exists. Note that Dynamic Data uses the `"_Edit"` suffix for Edit templates and no suffix for the `ReadOnly` templates. The Insert mode is even more specific because it might require special logic to deal with rows that have not been submitted to the database, and Dynamic Data chooses the template called `DateTime_Insert.ascx` if it exists.

Mode Fallback

If there is no matching field template for a particular control mode, Dynamic Data uses fallback logic to find another template for a more generic mode. In this ongoing example, when looking for an Insert template for the ShippedDate column, Dynamic Data will first try to find the template called `DateTime_Insert.ascx`. Because there is no field template with this name, it falls back from the Insert to the Edit mode and ends up using the template called `DateTime_Edit.ascx`, assuming it can handle both Edit and Insert modes. Similarly, if an Edit mode template cannot be found, Dynamic Data falls back to Read-only mode.

However, before Dynamic Data falls back to the next *mode*, it performs the *type* fallback. Consider UnitsInStock, an Int16 column from the Products table. When looking for an Edit template for this field, it tries using `System.Int16_Edit.ascx` and `Int16_Edit.ascx` templates first, but because they do not exist, it falls back to type `Int32` and checks for `System.Int32_Edit.ascx` and `Int32_Edit.ascx` and eventually finds `Integer_Edit.ascx`. If Dynamic Data performed the mode fallback first, it would skip this template and try to look for a ReadOnly template called `System.Int16.ascx` instead.

Navigation Columns

So far, our discussion has been about the lookup rules Dynamic Data uses for columns of *primitive* types that can be stored using built-in system types like `Int32` and `String`. The Dynamic Data metadata API represents them using the `MetaColumn` class.

Navigation columns, on the other hand, are columns that provide access to one or more entities that are defined as custom classes in the data model itself. There are three types of navigation columns from the Dynamic Data prospective:

- `MetaForeignKeyColumn` represents a navigation property that returns a single entity referenced by the underlying foreign key field. In the Northwind example, Product is a foreign key column of the Order_Detail entity that represents the Product entity referenced by the `ProductID` foreign key field.
- `MetaChildrenColumn` is an opposite of a foreign key. In the Northwind example, Order_Details column of the Product entity returns all `Order_Detail` items that reference a particular product instance.
- Many-to-Many column is a special case of a `MetaChildrenColumn` where the `IsManyToMany` property is set to `true`. In the Northwind example, the Territories column of the Employees entity returns all territories assigned to a particular employee. Likewise, the Employees column of the Territory entity returns all employees assigned to work in a particular Territory. Dynamic Data considers both of these columns as Many-to-Many.

Navigation columns get special treatment during field template lookup. Unless the developer already placed a `UIHintAttribute` on a navigation property, Dynamic Data uses the rules shown in Table 3.3 to come up with a default UI Hint.

TABLE 3.3 Navigation Column Rules

Column Type	UI Hint
`MetaForeignKeyColumn`	`ForeignKey`
`MetaChildrenColumn`	`Children`
`MetaChildrenColumn, IsManyToMany = true`	`ManyToMany`

In this example, it means that for the Product column of the Order_Details table, Dynamic Data uses a template called `ForeignKey.ascx`; for the Order_Details column of the Product table, it uses a template called `Children.ascx`; and for both Employee.Territories and Territory.Employees columns, it uses the template called `ManyToMany.ascx`.

Effectively, there is no type fallback during template lookup for navigation columns. However, Dynamic Data does perform mode fallback, so if your page needs a foreign key template in Insert mode, it first tries `ForeignKey_Insert.ascx`, then `ForeignKey_Edit.ascx`, and then `ForeignKey.ascx` if necessary. The default Dynamic Data project template

actually takes advantage of this to disable user interface for the children columns in Insert mode. If you open the `Children_Insert.ascx` field template, you see that it is empty. If this empty field template did not exist, `Children.ascx` would be used in its place, causing runtime errors because this template is not prepared to handle the scenario when the parent row does not exist.

Field Template Lookup Algorithm

Although the lookup rules for field templates might seem complex at a first, they are based on common-sense logic, and when you understand the idea behind them, the results will feel intuitive. Just remember that Dynamic Data tries to choose the most specific field template based on the column definition and control mode, and if such a field template does not exist, it gradually relaxes the matching rules until it finds a match or reports an error. The resulting behavior allows you to easily add new field templates where unique behavior is required to support a particular column type while allowing you to reuse existing templates with common behavior such as displaying read-only field values. Figure 3.4 shows the actual algorithm used by Dynamic Data. `FieldTemplateFactory`, a special class defined in the `System.Web.DynamicData` namespace, performs this task.

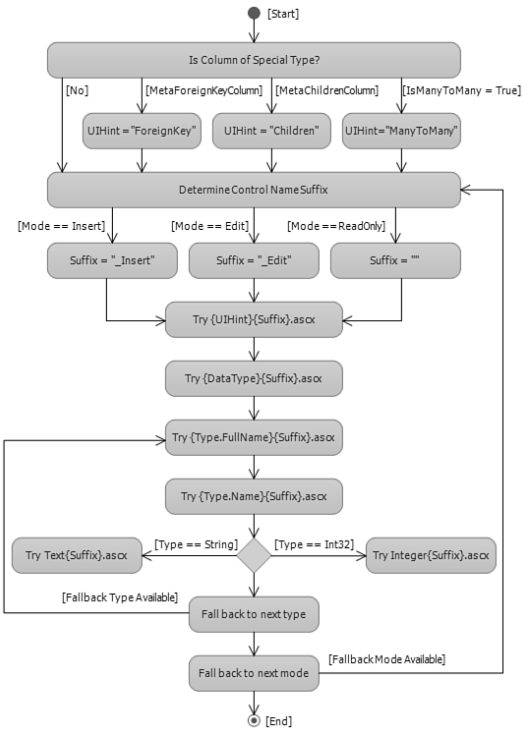


FIGURE 3.4 Field template lookup algorithm.

As an example, consider Table 3.4, which shows the sequence of template file names Dynamic Data will try when looking for a matching field template for the `HomePhone` property of the Employee entity.

TABLE 3.4 Example of Field Template Lookup for String Column

Mode="Insert"	**Mode="Edit"**	**Mode="ReadOnly"**
PhoneNumber_Insert.ascx	**PhoneNumber_Edit.ascx**	PhoneNumber.ascx
System.String_Insert.ascx	System.String_Edit.ascx	System.String.ascx
String_Insert.ascx	String_Edit.ascx	String.ascx
Text_Insert.ascx	Text_Edit.ascx	**Text.ascx**
PhoneNumber_Edit.ascx	PhoneNumber.ascx	
System.String_Edit.ascx	System.String.ascx	
String_Edit.ascx	String.ascx	
Text_Edit.ascx	Text.ascx	
PhoneNumber.ascx		
System.String.ascx		
String.ascx		
Text.ascx		

Because the HomePhone property has no `UIHintAttribute` and `UIHint` property is not specified for `DynamicControl` or `DynamicField` in page markup, the search sequence does not include the file names based on the `UIHint`. However, the HomePhone property does have the `DataType` specified in the data model using the `[DataType(DataType.PhoneNumber)]` attribute.

In Insert mode, the `FieldTemplateFactory` checks to see if `PhoneNumber_Insert.ascx`, `System.String_Insert.ascx`, `String_Insert.ascx`, and `Text_Insert.ascx` field templates exist before finding the `PhoneNumber_Edit.ascx` template created earlier in this chapter. If this template didn't exist, the `FieldTemplateFactory` would continue checking for `System.String_Edit.ascx`, `String_Edit.ascx`, and eventually find the `Text_Edit.ascx` field template provided by Dynamic Data projects out of the box.

In Edit mode, the `FieldTemplateFactory` starts with `PhoneNumber_Edit.ascx` and because it exists, uses it immediately. Likewise, had this field template not been defined, it would continue the search and eventually find the `Text_Edit.ascx` template provided by Dynamic Data.

In ReadOnly mode, the `FieldTemplateFactory` starts with `PhoneNumber.ascx`. Because the read-only version of the phone number field template does not exist, it continues by trying `System.String.ascx` and `String.ascx` and finally finds the `Text.ascx` field template provided by Dynamic Data out of the box.

Table 3.5 shows another example of field template lookup sequence, this time for the `UnitsInStock` property of the Product entity. This column is of type `Int32` and, unlike the HomePhone in the previous example, doesn't have a `DataTypeAttribute` applied to it.

Instead, this field template takes advantage of the fallback logic Dynamic Data uses for types and modes.

TABLE 3.5 Example of Field Template Lookup for Integer Column

Mode="Insert"	**Mode="Edit"**	**Mode="ReadOnly**
System.Int32_Insert.ascx	System.Int32_Edit.ascx	System.Int32.ascx
Int32_Insert.ascx	Int32_Edit.ascx	Int32.ascx
Integer_Insert.ascx	**Integer_Edit.ascx**	Integer.ascx
System.String_Insert.ascx	System.String_Edit.ascx	System.String.ascx
String_Insert.ascx	String_Edit.ascx	String.ascx
Text_Insert.ascx	Text_Edit.ascx	**Text.ascx**
System.Int32_Edit.ascx	System.Int32.ascx	
Int32_Edit.ascx	Int32.ascx	
Integer_Edit.ascx	Integer.ascx	
System.String_Edit.ascx	System.String.ascx	
String_Edit.ascx	String.ascx	
Text_Edit.ascx	Text.ascx	
System.Int32.ascx		
Int32.ascx		
Integer.ascx		
System.String.ascx		
String.ascx		
Text.ascx		

In Insert mode, the `FieldTemplateFactory` starts with the most specific template name (`System.Int32`) and mode (Insert). It tries `System.Int32_Insert`, `Int32_Insert`, and `Integer_Insert` before falling back to the next type—`String`. It continues trying `System.String_Insert`, `String_Insert`, and `Text_Insert` before falling back to the next mode, Edit, and repeating the steps starting with the most specific template name, `System.Int32`. Eventually, it finds `Integer_Edit.ascx`, the built-in field template provided by Dynamic Data for integer columns.

NOTE

For the UnitsInStock column, Dynamic Data uses both type aliases during the lookup sequence—`Integer`, which represents `Int32`, and `Text`, which represents `String`.

The Edit mode lookup sequence for the UnitsInStock column is very similar to that of Insert mode's. It starts with `System.Int32_Edit` and finds the built-in `Integer_Edit` template. The Read-only mode is different, however. Because Dynamic Data doesn't provide a built-in read-only template for integer columns, the `FieldTemplateFactory` falls back from `Integer` to `String` data type and eventually finds `Text.ascx`, a read-only template that can display field values of all types.

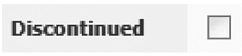
NOTE

Although the field template lookup algorithm might seem complex and require numerous file existence checks, in reality, Dynamic Data performs it only when a field template is needed for a particular column the first time. The results are stored in a hash table, and all subsequent lookups for this column are memory-based and very fast.

Built-in Field Templates

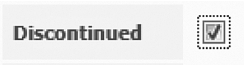
Dynamic Data web project templates in Visual Studio 2010 include a large number of ready-to-use field templates.

Boolean



The `Boolean.ascx` field template is used for Boolean columns in Read-only mode. It contains a disabled `CheckBox` control, checked when the field value is `true`.

Boolean_Edit



The `Boolean_Edit.ascx` field template is used for Boolean columns in Edit and Insert modes. It contains a `CheckBox` control users can use to set field value to `true` or `false`. This field template does not support null values and, unlike other well-behaved field templates, does not perform validation.

Children

The `Children.ascx` field template is used in Read-only and Edit modes for navigation properties that represent the many side of one-to-many relationships between entities. For example, the Customer entity in the Northwind sample data model has a children navigation property called `Orders`, which returns a collection of all orders placed by a given customer. In Read-only and Edit modes, the `Children.ascx` entity template displays a hyperlink to the dynamic `List.aspx` page that displays all child rows of the parent entity.

Children_Insert

The `Children_Insert.ascx` field template is empty. Because the dynamic `List.aspx` page cannot display child rows of a parent entity that has not been submitted to the database, this template prevents the `Children.ascx` field template from being used in Insert mode.

DateTime

The `DateTime.ascx` field template is used in Read-only mode to display `DateTime` field values. This field template contains a `Literal` control that injects the field value directly into the web page without any adorning HTML elements. This template is nearly identical to the `Text.ascx` template described in the following sections and could be removed, making `Text.ascx` responsible for displaying `DateTime` field values in Read-only mode as well.

DateTime_Edit

The `DateTime_Edit.ascx` field template is used for DateTime columns in Edit and Insert mode. It contains a `TextBox` control, accompanied by a `CustomValidator`, used to make sure that the value entered in the `TextBox` is a valid `DateTime`. It also contains a `RequiredFieldValidator`, used to enforce values for columns marked with the `RequiredAttribute`, a `RegularExpressionValidator`, used to enforce regular expression patterns for columns marked with the `RegularExpressionAttribute` and a `DynamicValidator`, used to enforce all other validation attributes applied to this column.

Decimal_Edit

The `Decimal_Edit.ascx` field template is used for `Decimal`, `Float`, and `Double` columns in Edit and Insert mode. It contains a `TextBox` control, accompanied by a `CompareValidator`, used to ensure that values entered in the `TextBox` are valid numbers. It also contains a `RequiredFieldValidator`, used to enforce values for columns marked with the `RequiredAttribute`; a `RangeValidator`, used to enforce range of values for columns marked with the `RangeAttribute`; a `RegularExpressionValidator`, used to enforce regular expression patterns for columns marked with the `RegularExpressionAttribute`; and a `DynamicValidator`, used to enforce all other validation attributes applied to this column.

EmailAddress

The `EmailAddress.ascx` field template is a read-only template used for `String` columns marked with the `DataType(DataType.EmailAddress)` attribute. It includes a `HyperLink` control that displays the field value as a link with the URL scheme `"mailto:"`. When the user clicks this link, most web browsers launch the default e-mail client installed on the computer and create a new message with the field value automatically added in the "To" list.

Enumeration

The `Enumeration.ascx` field template is used for enumeration columns in Read-only mode. It contains a `Literal` control that displays the underlying integral field value converted to its equivalent enumerated value and then converted to a `String`.

NOTE

Enumeration columns represent properties with an actual enumerated .NET data type (supported only by LINQ to SQL today) or integer properties that have been marked with the `EnumDataTypeAttribute` (supported by both LINQ to SQL and Entity Framework). As an example, consider the following enumerated type that could be used to track status of Orders in the Northwind database:

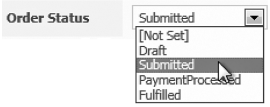
```
public enum OrderStatus: byte
{
  Draft = 0,
  Submitted = 1,
  PaymentProcessed = 2,
  Fulfilled = 3
}
```

In the sample data model, it could be expressed as an integer property called `OrderStatus` in the Order entity:

```
[EnumDataType(typeof(OrderStatus))]
public byte OrderStatus { get; set; }
```

By marking the `OrderStatus` property with the `EnumDataTypeAttribute`, you can make Dynamic Data treat it as enumeration instead of an integer and use the Enumeration.ascx field template, which displays `OrderStatus` field values as "`Draft`" and "`Submitted`," instead of "`1`" and "`2`."

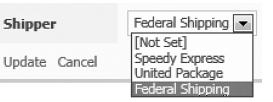
Enumeration Edit



The `Enumeration_Edit.ascx` field template is used for enumeration columns in Edit and Insert modes. It contains a `DropDownList` control, populated with enumerated items converted to string, plus "[Not Set]" if the column allows null values. This field template also includes a `RequiredFieldValidator` to enforce values for columns marked with the `RequiredAttribute` and a `DynamicValidator` to enforce all other validation attributes applied to this column.

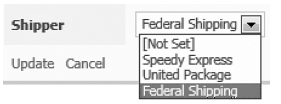
NOTE

When converting enumeration values to string, the Enumeration field templates do not automatically separate words concatenated to form a valid programmatic identifier. For example, an enumerated value `ReadOnly` would be normally displayed as "read-only" in a hand-coded web page. To make display names of the enumeration values more user-friendly, you can modify the Enumeration field templates to implement appropriate logic based either on the naming convention used to define the enumerated values, commonly PascalCase in .NET programs. Alternatively, you can use metadata attributes, such as `DisplayAttribute`, that can be applied to the enumerated values to specify the exact display string. Chapter 11, "Building Dynamic Forms," shows how to implement this.



ForeignKey

The `ForeignKey.ascx` is used for foreign key columns in Read-only mode. It contains a `HyperLink` control that displays the field value of the "display" column of the parent table. You can specify the display column explicitly by applying the `DisplayColumnAttribute` to the parent entity class, or it can be selected automatically based on the Dynamic Data heuristics. Users can click the link generated by the control to navigate to the `List.aspx` page of the parent table.



ForeignKey_Edit

The `ForeignKey_Edit.ascx` field template is used for foreign key columns in Edit and Insert modes. It contains a `DropDownList` control that displays all rows from the parent table. The display column of the parent table determines the `Text` of each `ListItem`, and its primary key columns determine its `Value`.

Integer_Edit

The `Integer_Edit.ascx` field template is used for `Byte`, `Int16`, `Int32`, and `Int64` columns in Edit and Insert modes. It contains a `TextBox` control, accompanied by a `CompareValidator`, which is used to ensure that values entered in the `TextBox` are valid numbers. It also contains a `RequiredFieldValidator`, used to enforce values for columns marked with the `RequiredAttribute`; a `RangeValidator`, used to enforce range of values for columns marked with the `RangeAttribute`; a `RegularExpressionValidator`, used to enforce regular expression patterns for columns marked with the `RegularExpressionAttribute`; and a `DynamicValidator`, used to enforce all other validation attributes applied to the column.

ManyToMany

The `ManyToMany.ascx` field template is used for children columns in a many-to-many relationship, such as the Employees column of the Territory entity in the Northwind sample database. This template contains a `Repeater` control that generates a `DynamicHyperLink` for every child entity. On the actual page, this looks like a list of hyperlinks that the user can click to navigate to the `Details.aspx` page of a particular child entity.

NOTE

Many-to-many relationships are natively supported in Entity Framework but not in LINQ to SQL. Therefore, only the Entity Framework flavors of the Dynamic Data project templates in Visual Studio include the `ManyToMany` field template. Unlike other field templates, which work equally well with both data access frameworks, the `ManyToMany` field template uses Entity Framework APIs directly.

LINQ to SQL natively supports only one-to-many relationships and requires you to have an explicit many-to-many entity in the model. However, it is possible to mimic many-to-many relationships by implementing a simple coding pattern as Eric Smith and Shannon Davidson describe in the article published on the Code Project, which can be found at the following location online:

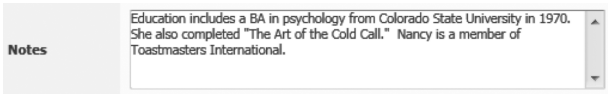
`http://www.codeproject.com/KB/linq/linq-to-sql-many-to-many.aspx`

It is possible to create Dynamic Data many-to-many field templates for LINQ to SQL based on this approach. However, this exercise is outside the scope of this chapter.

ManyToMany_Edit

The `ManyToMany_Edit.ascx` field template is used for many-to-many children columns in Edit and Insert modes. This template contains a `CheckBoxList` control that generates a check box for every entity the user can check to add it to the list of children.

MultilineText_Edit



The `MultilineText_Edit.ascx` field template is used for long string columns in Edit and Insert modes. It contains a `TextBox` control with `TextMode` property set to `MultiLine`. It also contains a `RequiredFieldValidator`, used to enforce values for columns marked with the `RequiredAttribute`, a `RegularExpressionValidator`, used to enforce regular expression patterns for columns marked with the `RegularExpressionAttribute` and a `DynamicValidator`, used to enforce all other validation attributes applied to the column.

String columns are considered *long* when they do not have maximum length specified in the data model, such as the Notes column of the Employee entity in the Northwind sample, which has type `NTEXT`. You can also force Dynamic Data to use the `MultilineText_Edit` field template for columns that do have maximum length by applying the `DataType(DataType.MultilineText)` attribute to their properties in the data model.

Text

Due to the data type fallback used during template lookup, the `Text.ascx` field template can be used in Read-only mode for columns of all primitive types with the exception of Boolean and DateTime columns, for which Dynamic Data provides specific templates. This field template includes a `Literal` control that displays the field value converted to a string.

Text_Edit

The `Text_Edit.ascx` field template is used for string columns in Edit and Insert modes. It includes a `TextBox` control, accompanied by a `RequiredFieldValidator`, used to enforce values for columns marked with the `RequiredAttribute`; a `RegularExpressionValidator`, used to enforce regular expression patterns for columns marked with the `RegularExpressionAttribute`; and a `DynamicValidator`, used to enforce all other validation attributes applied to the column.

Url

The `Url.ascx` field template is used for columns marked with the `DataType(DataType.Url)` attribute. It contains a `HyperLink` control that displays the field value as a link in the generated HTML. The `"http://"` scheme is automatically added to the URL unless the field value already contains it. Only HTTP and HTTPS schemes are recognized automatically. Although this field template does not require the column type to be only `String`, the field value converted to string must be a valid URL for the generated hyperlink to work correctly.

Summary

A field template is a special type of ASP.NET user control that inherits from the `FieldTemplateUserControl` base class provided by Dynamic Data. Multiple columns can reuse the same field template based on its compatibility with the column type and control mode. Changing a single field template automatically affects the user interface generated for all matching columns in the data model. Dynamic Data project template comes with a reasonable number of field templates that allow developers to create working web applications for many data types out of the box. Developers can easily modify the default field templates as well as add new ones to meet specific requirements of their projects.

Dynamic Data supports separate templates for Read-only, Edit, and Insert modes. Although a single template can support all three modes and programmatically display different versions of child controls, separating this logic into different templates helps to keep them simple and lightweight. Dynamic Data uses common-sense logic to find the best matching field template based on the column type and required control mode. If the most specific template does not exist, Dynamic Data gradually relaxes the matching rules and tries to find templates for a more generic column type and control mode.

Index

A

Action property, 180

Add New Item dialog, 6

advantages of Dynamic Data, 4

Aggregate method, 58

AggregateException, extending DynamicValidator to support, 254-255

AJAX (Asynchronous JavaScript and XML) Control Toolkit, 94

All method, 58

Any method, 58

applications

- creating, 4-6
- templates, 4-5
- Web Applications, 5-6
- Web Sites, 5

ASP.NET Dynamic Data Entities template, 4

ASP.NET Dynamic Data Entities Web Application template, 5

ASP.NET Dynamic Data LINQ to SQL template, 4

associations

- overview, 43-45
- querying across, 46

authorization rules for Web configuration files
limitations of, 461-462
overview, 458-460
AuthorizationAttribute, 477-480
MetaTable permission methods, implementing, 480-482
ObjectContext extended to authorize actions, 482-485
ObjectContext extended to enforce read permission, 485-486
automated testing, 230-231
different properties, testing validation of, 235-237
integration testing versus unit testing, 237-238
test isolation, 235
unit test, creating a, 231-234
AutoPostBack property, 337-338
Average method, 58
AzMan (Microsoft Authorization Manager), 463

B

backward compatibility, 4
BeforeSave methods, 276-277
Boolean Edit field template, 111
Boolean field template, 111
Boolean filter template, 151
Boolean.ascx field template, 111
Boolean_Edit.ascx field template, 111
built-in templates
entity templates, 134
field templates
Boolean Edit field template, 111
Boolean field template, 111
Children field template, 111
Children_Insert field template, 111
DateTime field template, 112
DateTime_Edit field template, 112
Decimal_Edit field template, 112
EmailAddress field template, 112
Enumeration Edit field template, 113
Enumeration field template, 112
ForeignKey field template, 114
ForeignKey_Edit field template, 114
Integer_Edit field template, 114
ManyToMany field template, 114
ManyToMany_Edit field template, 115
Multiline_Edit field template, 115
Text field template, 115
Text_Edit field template, 116
Url field template, 116
page templates, 154
Details page template, 155-159
Edit page template, 159-163
Insert page template, 169-171
List page template, 163-168
business logic, 263
at database level, 298-299
dynamic data applications, integrating database validation in, 299-303
stored procedures, implementing validation rules in, 306-308
triggers, implementing validation rules in, 304-305

default property values, 269-271
entity design, 264
conceptual model, 268-269
inheritance, impact on query performance of, 264-268
storage model, 268-269
entity interaction rules, 275
as context methods, 278-279
as Entity methods, 277
in BeforeSave methods, 276-277
property interaction rules, 272
ISavableObject interface, 273-275
OnPropertyChanging partial methods, 273
saving changes, 292-293
support, 208-209
validation rules, 280
business rule methods, reusing entity validation in, 290-291
deletion rules, 289-290
designing, 281-287
persistence validation, 280
related entities, 288
transition validation, 280
Web applications, invoking business rules in, 294-297

business rule methods, reusing entity validation in, 290-291

C

Cast method, 58

Children field template, 111

Children.ascx field template, 111

Children_Insert field template, 111

Children_Insert.ascx field template, 111

class inheritance, 72-73

class-level validation methods, 248-249

code generation
customizing, 227
context base class, customizing text template to change, 228
static constructors, customizing text template to generate, 230
text template used to replace built-in code generator, 227

ColumnProvider class, 435

columns
grid columns, 22-24
sorting across related entities, 400-401

compiled queries with LINQ to Entities, 55-57

composite entity templates, 122-123

Concat method, 58

conceptual model, 68-69, 268-269

Contains method, 58

context base class, 228

context methods, 278-279

control mode
field template lookup, 106
rules, 106

controls
data source controls, 11
DynamicControl, 13-14
DynamicEntity, 18-19
DynamicField, 22
DynamicFilter, 25-26
DynamicHyperLink, 33

EntityDataSource, 11
GridView control, 22
QueryableFilterRepeater, 28-29
QueryExtender, 25-29
Count method, 58
CRUD (Create, Read, Update, and Delete) operations, 153. *See also* page templates
custom data types in dynamic forms, 368-369
custom entity templates, 310-312
dynamic label control, 315
DynamicControl, extending, 319-320
custom field templates, creating, 324-327
custom properties, 350
building dynamic pages with, 432-433, 442-446
sorting, 446
dynamic data extended to support sort expression metadata, 447-449
entity model, specifying sort expression in, 446
support for, 434-436, 439-442
custom search pages
building, 401-402
dynamic filters, overcoming limitations of, 403-405
replacement filter controls, using, 406-407
related entities, displaying information from, 396-401
related entities, filtering based on, 407-409
nested filter control, initializing, 409-411
parent filter template, using, 419
query logic, implementing, 411-418
CustomQueryAttribute, 487
MetaTable.GetQuery implemented with, 488
row-level security, extending ObjectContext to enforce, 489-492
CustomValidationAttribute, 245-249

D

data access support, 201
data annotations, 17-18
field template lookup, 104
MetaColumn class, 207-210
MetaTable class, 198-200
persistence validation rules implemented with, 281
rules, 104
security, 476
AuthorizationAttribute, 477-486
CustomQueryAttribute, 487-492
Data Annotations namespace, 218
data binding, 96-99
data models, 190
creating, 6-11
naming, 7
Northwind data model, 8-9
registering, 12, 192-193
templates, 7
data source controls, 11
Data Transfer Object (DTO), 50

database
business rules in, 298-299
dynamic data applications, integrating database validation in, 299-303
stored procedures, implementing validation rules in, 306-308
triggers, implementing validation rules in, 304-305
Northwind sample database, downloading, 7
DataContextType property, 201
DataModelProvider
overview, 440-441
replacing built-in, 441-442
DataTypeAttribute, 245
DateTime field template, 112
DateTime.ascx field template, 112
DateTime_Edit field template, 112
DateTime_Edit.ascx field template, 112
DBNull values, 259-261
Decimal_Edit field template, 112
Decimal_Edit.ascx field template, 112
declarative validation, 238
DataTypeAttribute, 245
EnumDataTypeAttribute, 244-245
RangeAttribute, 239-240
RegularExpressionAttribute, 242-243
RequiredAttribute, 238-239
StringLengthAttribute, 241
declaratively configuring field templates, 322-323
default field templates, overriding, 359
default property values, 269-271
DefaultAutoFieldGenerator class, 376-377
code-behind, 24
markup, 23
DefaultIfEmpty method, 58
DefaultValue property, 143
DefaultValueAttribute, 209
Default_Edit entity template
code-behind, 21
markup, 20
delete procedure, 81
deletion rules, entity, 289-290
Description annotation, 356
***Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma, Helm, Johnson, and Vlissides), 287**
Details page template, 155-159
display support, 209
DisplayAttribute, 351-352
DisplayFormatAttribute, 360-361
Distinct method, 58
domain logic, 263
at database level, 298-299
dynamic data applications, integrating database validation in, 299-303
stored procedures, implementing validation rules in, 306-308
triggers, implementing validation rules in, 304-305
default property values, 269-271
entity design, 264
conceptual model, 268-269
inheritance, impact on query performance of, 264-268
storage model, 268-269

entity interaction rules, 275
as context methods, 278-279
as Entity methods, 277
in BeforeSave methods, 276-277
property interaction rules, 272
ISavableObject interface, 273-275
OnPropertyChanging partial methods, 273
saving changes, 292-293
support, 208-209
validation rules, 280
business rule methods, reusing entity validation in, 290-291
deletion rules, 289-290
designing, 281-287
persistence validation, 280
related entities, 288
transition validation, 280
Web applications, invoking business rules in, 294-297

DTO (Data Transfer Object), 50

dynamic entity templates, 124-128, 383
multimode, 313-315
page templates, extending, 386
two-column entity template, 388-393
UIHint attribute, 384-385
UnleashedEntity control, 386-388

dynamic fields, 22-24

dynamic filters
overcoming limitations of, 403
UnleashedFilter control, 404
UnleashedFilterExpression control, 405
UnleashedFilterRepeater control, 405
UnleashedFilterTemplate control, 405
replacement filter controls, using, 406-407

dynamic forms, 350-352
custom data types, 368-369
dynamic list pages compared, 421
Enum properties, 361-367
Enumeration field templates, 362-367
field templates
configuring, 357-359
interaction in, 378-383
parameters, specifying, 370-377
labels, configuring appearance of, 352-353
Description annotation, 356
Prompt annotation, 354-355
UIHint annotation, 369-371
values, configuring appearance of, 360-361

dynamic hyperlinks, 182-183

dynamic label control, creating, 315

dynamic LINQ queries, 143-145, 418

dynamic list pages
custom properties
building with, 432-433, 442-446
sorting, 446-449
support for, implementing, 434-436, 439-442
dynamic forms compared, 421
filter lookup rules, extending, 422-423
FilterFactory class, extending, 424-426

MetaModel class, extending, 429-430
MetaTable class, extending, 427-429
metadata attributes, generating, 450-452
dynamic templates, 464-465
DynamicControl
extending, 319-320, 343-344
and field template sample
code-behind, 14
markup, 13
DynamicControl_Init event handler, 21
DynamicDataRouteHandler class, 33
DynamicEntity control, 18-19, 157, 161
DynamicField controls, 22, 375
DynamicFilter controls, 25-26, 146-148, 166
DynamicHyperLink control, 33
DynamicValidator control, 161-162, 173, 220

E

eager loading in LINQ to Entities, 49
Edit field templates, 89-92
Edit mode templates
entity templates, 129-131
field templates, 89-91
Edit page template, 159-163
EDM (Entity Data Model), 66, 69-70
associations, 43-46
class inheritance, 72-73
conceptual model, 68-69
creating, 38-42
inheritance mapping, 73-74
mapping entity type to multiple tables, 71
mapping model, 70
storage model, 67
ElementAt method, 58
ElementAtOrDefault method, 58
EmailAddress field template, 112
EmailAddress.ascx field template, 112
encapsulation of field templates, 334
AutoPostBack property, 337
extending existing field templates, 337-341
FieldValue property, 336
FieldValueChanged event, 337
entities
deletion rules, 289-290
lifecycle, 60-66
mapping entity type to multiple tables, 71
modifying and saving, 46
optimistic concurrency control, 61-64
property interaction rules, 272
ISavableObject interface, 273-275
OnPropertyChanging partial methods, 273
read-only ObjectContext, 66
read-only, making state properties of entities, 281
related entities, validation of, 288
state changes, validating, 256
DBNull values, 259-261
ObjectStateManager used to access original property values, 256
validation methods, accessing to original property values in, 259

tracking changes, 60-61
transactions, 64-65
Entity Data Model (EDM), 66, 69-70
associations, 43-46
class inheritance, 72-73
conceptual model, 68-69
creating, 38-42
inheritance mapping, 73-74
mapping entity type to multiple tables, 71
mapping model, 70
storage model, 67
Entity Data Model Wizard, 8, 38-41
entity definition, 196-197
entity design, 264
conceptual model, 268-269
inheritance, impact on query performance of, 264
table-per-concrete type mapping, 265
table-per-hierarchy type mapping, 267-268
table-per-type type mapping, 266
storage model, 268-269
Entity Designer, 41
context base class, customizing text template to change, 228
default property values, 269-271
property visibility, changing, 358
static constructors, customizing text template to generate, 230
text template used to replace built-in code generator, 227
Entity Framework, 38
associations, 45
lifecycle of entities, 60
LINQ queries, 264
table-per-concrete type mapping, 265
table-per-hierarchy mapping, 267-268
table-per-type mapping, 266
LINQ to Entities, 47
compiled queries, 55-57
composing queries, 52-53
eager loading, 49
extension methods, 57-59
joins, 51-52
lazy loading, 49
object identity, 48
projections, 50
query execution, 47
sorting, 51
SQL translation, 53-55
metadata, 12
optimistic concurrency control, 61-64
read-only ObjectContext, 66
stored procedures, 74-77
delete procedure, 81
insert procedure, 78
update procedure, 79
tracking changes in entities, 60-61
transactions, 64-65
validation in, 221
code generation, customizing, 227-230
saving, validating entities before, 224-226
specifying validation attributes for entity classes, 222-223
TypeDescriptor, registering metadata provider with, 223

entity framework sample (code-behind), 10
entity framework sample (markup), 9
entity interaction rules, 275
as context methods, 278-279
as Entity methods, 277
in BeforeSave methods, 276-277
Entity methods, 277
entity templates, 18-22, 310
built-in entity templates, 134
composite entity templates, 122-123
creating, 120-121
custom entity templates, 310-312
dynamic label control, 315
DynamicControl, extending, 319-320
Default_Edit
code-behind, 21
markup, 20
dynamic entity templates, 124-128, 313, 383
multimode, 313-315
page templates, extending, 386
two-column entity template, 388-393
UIHint attribute, 384-385
UnleashedEntity control, 386-388
Edit mode templates, 129-131
field templates controlled by, 341-342
DynamicControl, extending, 343-344
UnleashedControl, 345-346
generic entity templates, 123-124
Insert mode templates, 129-131
lookup rules, 131
lookup algorithm, 133-134
template mode, 132
template name, 131
overview, 120
scaffolded properties, 127
EntityDataSource control, 11
EntityUIHintAttribute, 386
Enum properties, 361-367
EnumDataTypeAttribute, 244-245
Enumeration Edit field template, 113
Enumeration field templates, 112, 362-367
Enumeration filter template, 151
Enumeration.ascx field template, 112-113
Enumeration_Edit.ascx field template, 113
errors
handling, 296-297
messages, 219-221
multiple errors, reporting, 252-255
event handlers
DynamicControl_Init, 21
Label_Init, 21
Except method, 58
EXMX files, 67
extended field templates, 340
extending existing field templates, 338-341
extension methods with LINQ to Entities, 57-59
ExtractFieldValue method, 336

F

fallback logic, 105
field controls, 375
field templates, 13, 85-86
AutoPostBack property, setting, 338
base class, changing, 338

Boolean.ascx, 111
Boolean_Edit.ascx, 111
build-in field templates
  Boolean Edit field template, 111
  Boolean field template, 111
  Children field template, 111
  Children_Insert field template, 111
  DateTime field template, 112
  DateTime_Edit field template, 112
  Decimal_Edit field template, 112
  EmailAddress field template, 112
  Enumeration Edit field template, 113
  Enumeration field template, 112
  ForeignKey field template, 114
  ForeignKey_Edit field template, 114
  Integer_Edit field template, 114
  ManyToMany field template, 114
  ManyToMany_Edit field template, 115
  MultilineText_Edit field template, 115
  Text field template, 115
  Text_Edit field template, 116
  Url field template, 116
changing existing, 93-96
Children.ascx, 111
Children_Insert.ascx, 111
configuring, 321, 357
  declaratively, 322-323
  default field templates, overriding, 359
  programmatically, 324
  read-only, making properties, 358
  specifying a field template, 321-322
creating, 99-103
custom field templates, creating, 324-327
data binding, 96-99
DateTime.ascx, 112
DateTime_Edit.ascx, 112
Decimal_Edit.ascx, 112
dynamic forms, interaction in, 378, 381-383
Edit field templates, 89-92
edit mode templates, 89-91
EmailAddress.ascx, 112
encapsulation of field templates, improving, 334
  AutoPostBack property, 337
  extending existing field templates, 337-341
  FieldValue property, 336
  FieldValueChanged event, 337
entity templates used to control, 341-342
  DynamicControl, extending, 343-344
  UnleashedControl, 345-346
Enumeration field templates, 362-367
Enumeration.ascx, 112-113
Enumeration_Edit.ascx, 113
extended field templates, 340
field value display format, 87-89
field value edit format, 93
FieldValue property changing, updating data control when, 340
FieldValueChanged event, raising, 338
ForeignKey.ascx, 114
ForeignKey_Edit.ascx, 114

Insert field templates, 89-92
insert mode templates, 89
Integer_Edit
  code-behind, 15-16
  markup, 15
interaction between field templates, implementing, 327-329
  initial field values of other columns, accessing, 330
  modified field values of other columns, accessing, 331-332
  Region_Edit field template, 332
lookup rules, 104
  control mode, 106
  data annotations, 104
  field template lookup algorithm, 108-110
  lookup algorithm, 108-110
  mode fallback, 106
  navigation columns, 107-108
  type aliases, 105
  type fallback, 105
  type names, 104
ManyToMany.ascx, 114-115
ManyToMany_Edit.ascx, 115
MultilineText_Edit.ascx, 115
overview, 85
parameters, specifying, 370-377
parent field template, 397-400
Read-Only templates, 86-87
Region_Edit field template, 332
Text.ascx, 115
Text_Edit.ascx, 116
Url.ascx, 116

field value display format, 87-89
field value edit format, 93
fields
  dynamic fields, 22-24
  field templates, 13, 85-86
    Boolean.ascx, 111
    Boolean_Edit.ascx, 111
    changing existing field templates, 93-96
    Children.ascx, 111
    Children_Insert.ascx, 111
    creating, 99-103
    data binding, 96-99
    DateTime.ascx, 112
    DateTime_Edit.ascx, 112
    Decimal_Edit.ascx, 112
    Edit field templates, 89-92
    EmailAddress.ascx, 112
    Enumeration.ascx, 112-113
    Enumeration_Edit.ascx, 113
    field template sample, 13-14
    field value display format, 87-89
    field value edit format, 93
    ForeignKey.ascx, 114
    ForeignKey_Edit.ascx, 114
    Insert field templates, 89-92
    Integer_Edit field template, 15-16
    Integer_Edit.ascx, 114
    lookup rules, 104-110
    ManyToMany.ascx, 114-115
    ManyToMany_Edit.ascx, 115
    MultilineText_Edit.ascx, 115
    Read-Only templates, 86-87

Text.ascx, 115
Text_Edit.ascx, 116
Url.ascx, 116
FieldTemplateUserControl class, 324-327
FieldValue property, 336
changing, updating data control when, 340
ExtractFieldValue method, 336
FieldValueChanged event
overview, 337
raising, 338
filter lookup rules, extending, 422-423
FilterFactory class, extending, 424-426
MetaModel class, extending, 429-430
MetaTable class, extending, 427-429
filter templates, 25-29, 140
built-in filter templates
Boolean filter template, 151
Enumeration filter template, 151
ForeignKey filter template, 151
creating, 141-143
DefaultValue property, 143
dynamic LINQ queries, 143-145
GetQueryable method, 143
lookup rules, 150
OnFilterChanged method, 143
parent filter template
query logic in, 416-418
using, 419
search pages, building, 145-149
FilterFactory class, extending, 424-426
FindOtherFieldTemplate method, 381-383
First method, 58
FirstOrDefault method, 58
foreign key properties, 350
ForeignKey field template, 114
ForeignKey filter template, 27, 151
ForeignKey.ascx field template, 114
ForeignKey_Edit field template, 114
ForeignKey_Edit.ascx field template, 114
formatting options in MetaColumn class, 210
forms (dynamic), 350-352
custom data types, 368-369
Enum properties, 361-367
Enumeration field templates, 362-367
field template interaction in, 378-383
field templates, configuring, 357-359
labels, configuring appearance of, 352-356
UIHint annotation, 369-371
values, configuring appearance of, 360-361
FormView control, 170

G–H

Gamma, Eric (*Design Patterns: Elements of Reusable Object-Oriented Software*), 287
generated properties, 350
generic entity templates, 123-124
GetQueryable method, 143
GetRequestMetaTable method, 33
GetTable method, 194
global model registration, 195

Global.asax file
registering data model with, 12
registering URL routes in, 30
grid columns, 22-24
GridView control, 22, 164-166
group..by operator, 51
GroupBy method, 58

Helm, Richard (*Design Patterns: Elements of Reusable Object-Oriented Software*), 287

I

IDynamicValidatorException interface, 252
imperative validation, 245-249
informational messages, 305
inheritance mapping, 73-74
Insert field templates, 89-92
Insert mode templates
entity templates, 129-131
field templates, 89
Insert page template, 169-171
insert procedure, 78
Integer_Edit field template, 114
code-behind, 15-16
markup, 15
Integer_Edit.ascx field template, 114
integration tests, 237-238
Intersect method, 58
ISavableObject interface, 273-275
IsValid method, 217
IValidatableObject interface, 250-251

J–K–L

Johnson, Ralph (*Design Patterns: Elements of Reusable Object-Oriented Software*), 287
Join method, 58
Joins with LINQ to Entities, 51-52

keywords, partial, 18

Label_Init event handler, 21
lambda expressions, 139
Language Integrated Query. *See* LINQ
Last method, 58
LastOrDefault method, 59
lazy loading, 49, 288
LINQ (language-integrated query), 10, 264
associations, querying across, 46
dynamic LINQ queries, 143-145
expression mechanics, 411-415
filtering with, 139-140
simple queries, 43
table-per-concrete type mapping, 265
table-per-hierarchy mapping, 267-268
table-per-type mapping, 266
LINQ to Entities, 47
compiled queries, 55-57
composing queries, 52-53
eager loading, 49
extension methods, 57-59
joins, 51-52
lazy loading, 49
object identity, 48
projections, 50

query execution, 47
sorting, 51
SQL translation, 53-55
LINQ to Objects, 194
LINQ to SQL, 37, 115
List page template, 163-168
list pages (dynamic)
custom properties
building with, 432-433, 442-446
sorting, 446-449
support for, implementing, 434-436, 439-442
dynamic forms compared, 421
filter lookup rules, extending, 422-423
FilterFactory class, extending, 424-426
MetaModel class, extending, 429-430
MetaTable class, extending, 427-429
metadata attributes, generating, 450-452
LongCount method, 59
lookup algorithm
entity templates, 133-134
field templates, 108-110
lookup rules
entity templates, 131
lookup algorithm, 133-134
template mode, 132
template name, 131
field templates, 104
control mode, 106
data annotations, 104
lookup algorithm, 108-110
mode fallback, 106
navigation columns, 107-108
type aliases, 105
type fallback, 105
type names, 104
filter templates, 150
page templates, 185

M

ManyToMany field template, 114
ManyToMany.ascx field template, 114-115
ManyToMany_Edit field template, 115
ManyToMany_Edit.ascx field template, 115
mapping
table-per-concrete type mapping, 265
table-per-hierarchy type mapping, 267-268
table-per-type type mapping, 266
entity type to multiple tables, 71
model, 70
Max method, 59
meta model, 190
MetaChildrenColumn class, 213-214
MetaColumn class, 203, 355
business logic support, 208-209
data annotations, 207-210
display support, 209
formatting options, 210
property definition, 204-206
property kind, 206-207
scaffolding support, 208

metadata, 12. *See also* data annotations
attributes, generating, 450-452
sort expression through, 447-449
Metadata API, 190-191
MetaChildrenColumn class, 213-214
MetaColumn class, 203
business logic support, 208-209
data annotations, 207-210
display support, 209
formatting options, 210
property definition, 204-206
property kind, 206-207
scaffolding support, 208
MetaForeignKeyColumn class, 211-212
MetaModel class, 192
data model registration, 192-193
global model registration, 195
MetaModel objects, accessing, 193-194
scaffolding support, 196
MetaTable class, 196
data access support, 201
data annotations, 198-200
entity definition, 196-197
presentation logic support, 202-203
security support, 203
security, 462-463
dynamic templates modified to support, 464-465
page templates modified to hide inaccessible actions, 468-471
page templates modified to prevent unauthorized access, 465-468
metadata class, 222-223
metadata explorer, 191
MetaForeignKeyColumn class, 211-212
MetaModel class, 192
data model registration, 192-193
extending, 429-430
global model registration, 195
MetaTable objects, accessing, 193-194
scaffolding support, 196
MetaTable class, 196
data access support, 201
data annotations, 198-200
entity definition, 196-197
extending, 427-429
presentation logic support, 202-203
security support, 203
MetaTable objects, accessing, 193-194
MetaTable permission methods, implementing, 480-482
Microsoft Authorization Manager (AzMan), 463
Min method, 59
mode fallback
field template lookup, 106
rules, 106
model providers, 434
ColumnProvider class, 435
custom properties, building dynamic pages with, 442-446
DataModelProvider, 440-442
TableProvider class, 436, 439
MultilineText_Edit.ascx field template, 115

Multiline_Edit field template, 115
multiple rules, validating, 250-251
MVC (Model-View-Controller) framework, 347
MVC (Model-View-Controller) pattern, 346

N–O

naming data model files, 7
navigation columns, 107-108
nested filter control, initializing, 409-411
Northwind data model, 8-9
Northwind sample database, downloading, 7
NorthwindEntities class, 12
null values, 247-248

object identity, 48
ObjectContext, 43, 60-61
extended to authorize actions, 482-485
extended to enforce read permission, 485-486
multiple errors, modifying ObjectContext to report, 253-254
read-only, 66
ObjectStateManager, 256
OnFilterChanged method, 143
OnInit event handler, 470
OnPropertyChanging partial methods, 273
optimistic concurrency control, 61-64
orderby keyword, 50
OrderBy method, 59
OrderByDescending method, 59
original property values, validation methods for accessing, 259

P

page templates, 29-34
built-in page templates, 154
Details page template, 155-159
Edit page template, 159-163
Insert page template, 169-171
List page template, 163-168
creating, 172-173
custom page templates, 174-179
extending, 386
inaccessible actions, hiding, 468-471
lookup rules, 185
unauthorized access of, preventing, 465-468
URL routing, 180
Action route parameter, 180
dynamic hyperlinks, 182-183
route evaluation and constraints, 184
Table route parameter, 180
URL parameters, 181
ViewName route parameter, 185
parent field template, 397-400
parent filter template
query logic in, 416-418
using, 419
partial keyword, 18
persistence validation, 280
data annotations used to implement rules, 281
state-specific persistence validation in separate validation methods, implementing, 285-287

PlaceHolder control, 392

presentation logic, 202-203, 211-212

primary keys, 206

programmatically configuring field templates, 324

project templates, 4-5

projections in LINQ to Entities, 50

Prompt annotation, 354-355

property definition in MetaColumn class, 204-206

property interaction rules, 272

ISavableObject interface, 273-275

OnPropertyChanging partial methods, 273

property kind in MetaColumn class, 206-207

provider model, 190

Q–R

query execution in LINQ to Entities, 47, 52-53

query logic, implementing, 411-418

Query parameters, URL parameters compared, 181

QueryableFilterRepeater control, 28-29, 148-149

QueryExtender control, 157

DynamicFilter sample, 25-26

QueryableFilterRepeater sample, 28-29

RangeAttribute, 239-240

Read-Only field templates, 86-87

read-only ObjectContext, 66

read-only templates, 86

read-only, making state properties of entities, 281

Region_Edit field template, 332

registering

data models, 12

URL routes, 30

RegularExpressionAttribute, 242-243

related entities

column sorting, 400-401

displaying information from, 396-400

filtering based on, 407-409

nested filter control, initializing, 409-411

parent filter template, using, 419

query logic, implementing, 411-418

validation rules, 288

RequiredAttribute, 238-239

Reverse method, 59

route constraints, 184, 457-458

route evaluation and constraints, 184

routing, URL, 29-34

row-level security

ObjectContext extended to enforce, 489-492

overview, 472-475

rules, validating, 250-255

S

SaveChanges method, 46, 225-227, 292

saving

validating entities before, 224-226

changes in business rule methods, 292-293

scaffolded properties, 127

scaffolding

overview, 456-457

support

MetaColumn class, 208

MetaModel class, 196

search functionality, 137-140. *See also* filter templates, 137

search pages

custom search pages

building, 401-407

related entities, displaying information from, 396-401

related entities, filtering based on, 407-419

filter templates used to build, 145-149

security, 455-456

annotations, 476

AuthorizationAttribute, 477-486

CustomQueryAttribute, 487-492

metadata API, 462-463

dynamic templates modified to support, 464-465

page templates modified to hide inaccessible actions, 468-471

page templates modified to prevent unauthorized access, 465-468

route constraints, 457-458

row-level, 472-475, 489-492

scaffolding, 456-457

Web.config authorization rules, 458-462

support, 203

trimming, 462

Select method, 59

SelectMany method, 59

SequenceEqual method, 59

simple queries with LINQ, 43

Single method, 59

SingleOrDefault method, 59

Skip method, 59

SkipWhile method, 59

Sorting with LINQ to Entities, 51

SQL translation with LINQ to Entities, 53-55

state transitions and validation in business rule methods, implementing, 282-285

state-specific persistence validation in separate validation methods, implementing, 285-287

static constructors, 230

static LINQ queries, 418

storage model, 67, 268-269

stored procedures, 74-77

delete procedure, 81

implementing validation rules in, 306-308

insert procedure, 78

update procedure, 79

StringLengthAttribute, 241

Sum method, 59

system errors, 305

System.ComponentModel.DataAnnotations namespace, 238, 250

T

Table property, 180

table provider, 197

table-per-concrete type mapping, 73, 265

table-per-hierarchy type mapping, 74, 267-268

table-per-type mapping, 73, 266

TableProvider class, 436, 439

tables

- inheritance mapping, 73-74
- mapping entity type to multiple tables, 71
- scaffolding, 456-457

Take method, 59

TakeWhile method, 59

template controls, 375

template mode rules, 132

template name rules, 131

templates, 4-5

- data model templates, 7
- entity templates, 18-22, 310
  - built-in entity templats, 134
  - composite entity templates, 122-123
  - creating, 120-121
  - custom entity templates, 310-312, 315, 319-320
  - Default_Edit, 20-21
  - Edit mode templates, 129-131
  - field templates controlled by, 341-346
  - generic entity templates, 123-124
  - Insert mode templates, 129-131
  - lookup rules, 131-134
  - overview, 120
  - scaffolded properties, 127
- field templates, 13, 85-86
  - AutoPostBack property, setting, 338
  - base class, changing, 338
  - Boolean.ascx, 111
  - Boolean_Edit.ascx, 111
  - built-in field templates, 111-116
  - changing existing field templates, 93-96
  - Children.ascx, 111
  - Children_Insert.ascx, 111
  - configuring, 321-324, 357-359
  - creating, 99-103
  - custom field templates, creating, 324-327
  - data binding, 96-99
  - DateTime.ascx, 112
  - DateTime_Edit.ascx, 112
  - Decimal_Edit.ascx, 112
  - Edit field templates, 89-92
  - edit mode templates, 89-91
  - edit templates, 93
  - EmailAddress.ascx, 112
  - encapsulation of field templates, improving, 334-341
  - entity templates used to control, 341-346
  - Enumeration.ascx, 112-113
  - Enumeration_Edit.ascx, 113
  - extended field templates, 340
  - field template sample, 13-14
  - field value display format, 87-89
  - field value edit format, 93
  - FieldValue property changing, updating data control when, 340

FieldValueChanged event, raising, 338
ForeignKey.ascx, 114
ForeignKey_Edit.ascx, 114
Insert field templates, 89-92
insert mode templates, 89
Integer_Edit field template, 15-16
Integer_Edit.ascx, 114
interaction between field templates, implementing, 327-332
lookup rules, 104-110
ManyToMany.ascx, 114-115
ManyToMany_Edit.ascx, 115
MultilineText_Edit.ascx, 115
overview, 85
parameters, specifying, 370-377
Read-Only templates, 86-87
Text.ascx, 115
Text_Edit.ascx, 116
Url.ascx, 116
filter templates, 25-29, 140
built-in filter templates, 151
creating, 141-143
DefaultValue property, 143
dynamic LINQ queries, 143-145
GetQueryable method, 143
lookup rules, 150
OnFilterChanged method, 143
search pages, building, 145-149
page templates, 29-34
built-in page templates, 154-171
creating, 172-173
custom page templates, 174-179
lookup rules, 185
URL routing, 180-185
project templates, 4-5
testing (automated), 230-231
different properties, testing validation of, 235-237
integrated testing versus unit testing, 237-238
test isolation, 235
unit test, creating a, 231-234
Text field template, 115
text template
code generator replaced with, built-in, 227
context base class, customizing text template to change, 228
static constructors, customizing text template to generate, 230
Text.ascx field template, 115
Text_Edit field template, 116
Text_Edit.ascx field template, 116
ThenBy method, 59
ThenByDescending method, 59
ToString method, 200
tracking changes, 60-61
transactions
managing, 293
overview, 64-65
TransactionScope, 234, 238, 293
transition validation, 280
triggers, implementing validation rules in, 304-305
two-column entity template, 388-393

type aliases, 105
type fallback, 105
type names, 104
TypeDescriptor class, 223

U

UIHint annotation, 369-371
UIHint attribute, 384-385
UIHint property, 386
Union method, 59
unit tests
integration test compared, 237-238
overview, 231-234
UnleashedControl, 345-346, 372-375
UnleashedEntity control, 386-388
UnleashedFilter control, 404
UnleashedFilterExpression control, 405
UnleashedFilterRepeater control, 405
UnleashedFilterTemplate control, 405
UnleashedLabel control, 316-318
update procedure, 79
Update Wizard, 44
Url field template, 116
URL parameters, Query parameters compared, 181
URL routing, 29-34
Action route parameter, 180
dynamic hyperlinks, 182-183
route evaluation and constraints, 184
Table route parameter, 180
URL parameters, 181
ViewName route parameter, 185
Url.ascx field template, 116
user errors, 305

V

ValidateException method, 255
ValidateObject method, 218-219
validation, 217-219
automated testing, 230-231
different properties, testing validation of, 235-237
integration testing versus unit testing, 237-238
test isolation, 235
unit test, creating a, 231-234
class-level validation methods, 248-249
Data Annotations namespace, 218
declarative, 238
DataTypeAttribute, 245
EnumDataTypeAttribute, 244-245
RangeAttribute, 239-240
RegularExpressionAttribute, 242-243
RequiredAttribute, 238-239
StringLengthAttribute, 241
dynamic data applications, integrating database validation in, 299-303
entity state changes, 256
DBNull values, 259-261
ObjectStateManager used to access original property values, 256

validation methods, accessing to original property values in, 259
error messages, 219-221
imperative, 245-249
in Entity Framework, 221
code generation, customizing, 227-230
saving, validating entities before, 224-226
specifying validation attributes for entity classes, 222-223
TypeDescriptor, registering metadata provider with, 223
multiple rules, 250-255
null values, 247-248
validation rules, 280
business rule methods, reusing entity validation in, 290-291
deletion rules, 289-290
designing, 281
data annotations, implementing persistence validation rules with, 281
read-only, making state properties of entities, 281
state transitions and validation in business rule methods, implementing, 282-285
state-specific persistence validation in separate validation methods, implementing, 285-287
persistence validation, 280
related entities, 288
transition validation, 280
ValidationAttribute class, 217, 220
ValidationContext object, 218
ValidationResult, 246-247, 250-251
Validator class, 220, 252
values, configuring appearance of, 360-361
ViewName property, 185
Vlissides, John (*Design Patterns: Elements of Reusable Object-Oriented Software*), 287

W–Z

Web Application projects, 5-6
Web applications, invoking business rules in, 294-297
Web Site projects, 5
Web.config authorization rules, 458-462
Where method, 59, 139, 145
Zip method, 59