**Microsoft .NET RIA Services Overview**

July 2009 Preview

July 2009

Contents

[1. Introduction 11](#_Toc234924736)

[1.1 Challenges of using data in N-tier applications 11](#_Toc234924737)

[1.2 Value propositions 12](#_Toc234924738)

[1.3 Key concepts 13](#_Toc234924739)

[1.4 Roadmap 15](#_Toc234924740)

[1.5 New in July 2009 CTP 15](#_Toc234924741)

[2 Understanding N-tier Silverlight Application Projects 16](#_Toc234924742)

[2.1 Creating the project 16](#_Toc234924743)

[2.2 Converting an existing web application 18](#_Toc234924744)

[2.3 Exposing data from the mid-tier 19](#_Toc234924745)

[2.4 Accessing data from the mid-tier 19](#_Toc234924746)

[2.5 Creating a DomainService on the mid-tier 21](#_Toc234924747)

[2.6 Accessing the data from the Silverlight client 23](#_Toc234924748)

[2.7 Using the data on the Silverlight client 25](#_Toc234924749)

[2.8 Summary 26](#_Toc234924750)

[3 Understanding Silverlight Client Code Generation 28](#_Toc234924751)

[3.1 The sample application 28](#_Toc234924752)

[3.1.1 Sample City entity and store 29](#_Toc234924753)

[3.1.2 Sample DomainService 30](#_Toc234924754)

[3.1.3 The generated code 31](#_Toc234924755)

[3.1.4 Adding metadata to the entity 32](#_Toc234924756)

[3.1.5 Adding shared code 34](#_Toc234924757)

[3.1.6 The final generated code 35](#_Toc234924758)

[3.2 How it works 36](#_Toc234924759)

[3.2.1 The code-generation algorithms 36](#_Toc234924760)

[3.2.2 Generated entity proxy classes 36](#_Toc234924761)

[3.2.3 Generated custom attributes 37](#_Toc234924762)

[3.2.4 The metadata class 37](#_Toc234924763)

[3.2.5 The DomainContext class 38](#_Toc234924764)

[3.2.6 The shared file copy algorithms 38](#_Toc234924765)

[3.2.7 The automatic detection of shared code 39](#_Toc234924766)

[3.3 Summary 39](#_Toc234924767)

[3.4 Generated code 40](#_Toc234924768)

[4 How to Query Entities 44](#_Toc234924769)

[4.5 Query Methods 44](#_Toc234924770)

[4.6 Defining Query Methods 44](#_Toc234924771)

[4.7 Client Code Generation for Query Methods 45](#_Toc234924772)

[4.7.1 Using Generated Query Methods 46](#_Toc234924773)

[4.8 Using the IncludeAttribute 47](#_Toc234924774)

[4.8.1 Returning Related Entities 47](#_Toc234924775)

[4.8.2 Denormalizing Associated Data 48](#_Toc234924776)

[4.9 Query Operators 48](#_Toc234924777)

[4.10 POST support for Query operations 49](#_Toc234924778)

[4.11 Supported Types 49](#_Toc234924779)

[4.11.1 Simple Supported Types 49](#_Toc234924780)

[4.11.2 Complex Supported Types 50](#_Toc234924781)

[5 How to Modify Entities 51](#_Toc234924782)

[5.1 Sample Setup 51](#_Toc234924783)

[5.2 Updating entities 52](#_Toc234924784)

[5.2.1 DomainService Update Methods 52](#_Toc234924785)

[5.3 EntityContainer, EntityList and Change Tracking 54](#_Toc234924786)

[5.4 Change Sets (Unit of Work) 56](#_Toc234924787)

[5.4.1 Client Side Changeset Processing 56](#_Toc234924788)

[5.4.2 Server Side Changeset Processing 57](#_Toc234924789)

[5.5 Error Handling 58](#_Toc234924790)

[5.6 Concurrency Conflicts 59](#_Toc234924791)

[5.7 Transactions 59](#_Toc234924792)

[5.8 Identity Management 60](#_Toc234924793)

[5.9 IEditableObject/IChangeTracking/IEditableCollection 61](#_Toc234924794)

[5.9.1 IEditableObject 61](#_Toc234924795)

[5.9.2 IChangeTracking/IRevertibleChangeTracking 61](#_Toc234924796)

[5.9.3 IEditableCollection/IEditableCollection<T> 62](#_Toc234924797)

[6 Using Custom Methods and Service Operations 63](#_Toc234924798)

[6.1 Sample Setup 63](#_Toc234924799)

[6.2 Writing Application Logic 64](#_Toc234924800)

[6.2.1 Custom Methods 65](#_Toc234924801)

[6.2.2 Service Operations 65](#_Toc234924802)

[6.3 Conclusion 66](#_Toc234924803)

[7 Understanding Convention and Configuration 67](#_Toc234924804)

[7.1 Motivation behind convention-based design 67](#_Toc234924805)

[7.2 Rules that apply to all DomainOperations 67](#_Toc234924806)

[7.3 DomainOperations 67](#_Toc234924807)

[7.3.1 Insert 67](#_Toc234924808)

[7.3.2 Update 68](#_Toc234924809)

[7.3.3 Delete 68](#_Toc234924810)

[7.3.4 Query 68](#_Toc234924811)

[7.3.5 Custom 69](#_Toc234924812)

[7.3.6 Resolve 69](#_Toc234924813)

[7.3.7 ServiceOperation 69](#_Toc234924814)

[7.3.8 IgnoreOperationAttribute 69](#_Toc234924815)

[8 Asynchronous Domain Operations 71](#_Toc234924816)

[8.1 OperationBase 71](#_Toc234924817)

[8.2 LoadOperation 72](#_Toc234924818)

[8.3 SubmitOperation 73](#_Toc234924819)

[8.4 InvokeOperation 73](#_Toc234924820)

[9 Using the Silverlight DomainDataSource 74](#_Toc234924821)

[9.1 Sample Setup 74](#_Toc234924822)

[9.2 DomainDataSource 76](#_Toc234924823)

[9.2.1 Loading Data 76](#_Toc234924824)

[9.2.2 Parameterized Query Methods 76](#_Toc234924825)

[9.2.3 Shaping Data 77](#_Toc234924826)

[9.2.4 Editing 82](#_Toc234924827)

[9.3 Summary 82](#_Toc234924828)

[9.4 See Also 82](#_Toc234924829)

[9.5 Reference XAML 82](#_Toc234924830)

[10 How to use metadata classes 85](#_Toc234924831)

[10.1 Procedure 85](#_Toc234924832)

[10.1.1 Prerequisites: 85](#_Toc234924833)

[10.1.2 Steps (using Wizard): 85](#_Toc234924834)

[10.1.3 Steps (manual): 86](#_Toc234924835)

[10.2 Example 87](#_Toc234924836)

[10.3 External Metadata 88](#_Toc234924837)

[10.4 See Also 88](#_Toc234924838)

[11 How to Add Validation to Entities 89](#_Toc234924839)

[11.1 Procedure 89](#_Toc234924840)

[11.1.1 Prerequisites: 89](#_Toc234924841)

[11.1.2 Steps: 89](#_Toc234924842)

[11.1.3 Notes: 89](#_Toc234924843)

[11.2 Example 89](#_Toc234924844)

[11.3 Extensibility 96](#_Toc234924845)

[11.4 See Also 96](#_Toc234924846)

[12 How to Share Code Across Tiers 97](#_Toc234924847)

[12.1 Generated proxy types 97](#_Toc234924848)

[12.2 Shared source files 98](#_Toc234924849)

[12.2.1 Sharing files by naming pattern 98](#_Toc234924850)

[12.2.2 Sharing files by linking 100](#_Toc234924851)

[12.3 Shared class libraries 102](#_Toc234924852)

[12.4 Summary 103](#_Toc234924853)

[13 Using Authentication, Roles and Profiles 104](#_Toc234924854)

[13.1 Authentication 104](#_Toc234924855)

[13.1.1 Server Tasks 104](#_Toc234924856)

[13.1.2 Client Tasks 105](#_Toc234924857)

[13.1.3 Client Usage 106](#_Toc234924858)

[13.2 Roles 107](#_Toc234924859)

[13.2.1 Server Tasks 107](#_Toc234924860)

[13.2.2 Client Usage 107](#_Toc234924861)

[13.3 Profile 107](#_Toc234924862)

[13.3.1 Server Tasks 107](#_Toc234924863)

[13.3.2 Client Usage 108](#_Toc234924864)

[14 How to Restrict Access to DomainService Operations 110](#_Toc234924865)

[14.1 Procedure 110](#_Toc234924866)

[14.1.1 Prerequisites: 110](#_Toc234924867)

[14.1.2 Steps 110](#_Toc234924868)

[14.2 Example 110](#_Toc234924869)

[14.3 Extensibility 111](#_Toc234924870)

[14.4 See Also 112](#_Toc234924871)

[15 Inside the Business Application Template 113](#_Toc234924872)

[15.1 Getting Started 113](#_Toc234924873)

[15.1.1 Project Structure 113](#_Toc234924874)

[15.1.2 Controls 115](#_Toc234924875)

[15.2 .NET RIA Services Usage 117](#_Toc234924876)

[15.2.1 Forms Authentication 117](#_Toc234924877)

[15.2.2 Server 117](#_Toc234924878)

[15.2.3 Client 118](#_Toc234924879)

[15.2.4 Windows Authentication 119](#_Toc234924880)

[15.2.5 User Registartion Service 119](#_Toc234924881)

[16 Writing a https Enabled DomainService 120](#_Toc234924882)

[16.1 Brief Example 120](#_Toc234924883)

[16.2 Further Information 120](#_Toc234924884)

[17 Code Generated Hook Points 122](#_Toc234924885)

[17.1 Overview 122](#_Toc234924886)

[17.2 DomainContext Hookpoints 122](#_Toc234924887)

[17.2.1 OnCreated() 122](#_Toc234924888)

[17.3 Entity Hookpoints 122](#_Toc234924889)

[17.3.1 OnCreated() 122](#_Toc234924890)

[17.3.2 OnLoaded(bool isIntialLoad) 122](#_Toc234924891)

[17.3.3 On[PropertyName]Changing([PropetyType] value) 123](#_Toc234924892)

[17.3.4 On[PropertyName]Changed() 123](#_Toc234924893)

[17.3.5 On[CustomMethodName]Invoking 123](#_Toc234924894)

[17.3.6 On[CustomMethodName]Invoked 123](#_Toc234924895)

[17.3.7 Is[CustomMethodName]Invoked 124](#_Toc234924896)

[17.4 Scenarios 124](#_Toc234924897)

[17.4.1 DomainContext Initialization 124](#_Toc234924898)

[17.4.2 Entity Initialization 124](#_Toc234924899)

[17.4.3 125](#_Toc234924900)

[17.4.4 Computed properties 125](#_Toc234924901)

[18 How to add computed properties 126](#_Toc234924902)

[18.1 Sample Setup 126](#_Toc234924903)

[18.2 Adding client side computed property 126](#_Toc234924904)

[18.3 Data binding computed property to UI 127](#_Toc234924905)

[19 Understanding N-tier Class Libraries 128](#_Toc234924906)

[19.1 Overview 128](#_Toc234924907)

[19.1.1 Building N-tier applications using .NET RIA Services 128](#_Toc234924908)

[19.1.2 Using class libraries on the mid-tier prior to the July 09 preview 129](#_Toc234924909)

[19.1.3 New in July 09 Preview – the .NET RIA Services Class Library 130](#_Toc234924910)

[19.2 Using the .NET RIA Services Class Library 131](#_Toc234924911)

[19.2.1 Start with an existing Web application 131](#_Toc234924912)

[19.2.2 Create a new .NET RIA Services class library using the new template 131](#_Toc234924913)

[19.2.3 Add references to the class libraries 134](#_Toc234924914)

[19.2.4 Add a model to the mid-tier library 134](#_Toc234924915)

[19.2.5 Create a DomainService in the mid-tier library 135](#_Toc234924916)

[19.2.6 Build the solution 137](#_Toc234924917)

[19.2.7 Use the generated code from the client application 137](#_Toc234924918)

[19.3 Possible project structures 138](#_Toc234924919)

[19.3.1 Simplest application – no class libraries 138](#_Toc234924920)

[19.3.2 Mid-tier class libraries 139](#_Toc234924921)

[19.3.3 Using the new .NET RIA Services class libraries 141](#_Toc234924922)

[19.3.4 Effects of mixing the RIA Link between the Web project and class libraries 143](#_Toc234924923)

[19.3.5 View model class libraries 145](#_Toc234924924)

[19.3.6 Recommended project structures 145](#_Toc234924925)

[19.4 Miscellaneous 146](#_Toc234924926)

[19.4.1 What does the “Enable .NET RIA Services” checkbox do? 146](#_Toc234924927)

[19.4.2 Special issues when placing DAL models in class libraries 147](#_Toc234924928)

[19.4.3 Special issues when creating DomainServices in class libraries 147](#_Toc234924929)

[19.5 Summary 148](#_Toc234924930)

[20 How to Work with Multiple DomainContexts 149](#_Toc234924931)

[20.1 Procedure 149](#_Toc234924932)

[20.1.1 Prerequisites: 149](#_Toc234924933)

[20.1.2 Steps: 149](#_Toc234924934)

[20.2 Basic Example 150](#_Toc234924935)

[20.2.1 Server Side 150](#_Toc234924936)

[20.2.2 Client Project 151](#_Toc234924937)

[20.3 Advanced Example 151](#_Toc234924938)

[20.3.1 Client Project 152](#_Toc234924939)

[**20.4** **See Also** 153](#_Toc234924940)

[21 How to unit test business logic 154](#_Toc234924941)

[21.1 Sample Setup 154](#_Toc234924942)

[22 How to display localized error messages 162](#_Toc234924943)

[22.1 Sample Setup 162](#_Toc234924944)

[22.2 Creating error messages as resource file 162](#_Toc234924945)

[23 Using the ASP.NET DomainDataSource 164](#_Toc234924946)

[23.1 Prerequisites 164](#_Toc234924947)

[23.2 Procedures 164](#_Toc234924948)

[23.2.1 To create a Web application 164](#_Toc234924949)

[23.2.2 To add the database file to the Web application 164](#_Toc234924950)

[23.2.3 To create the data model using LINQ to SQL 165](#_Toc234924951)

[23.2.4 To create the domain data service 165](#_Toc234924952)

[23.2.5 To configure the domain data service 165](#_Toc234924953)

[23.2.6 To test the domain data source 166](#_Toc234924954)

[24 ADO.NET Data Services Integration 167](#_Toc234924955)

[24.1 The Sample Application 168](#_Toc234924956)

[24.2 Adding an ADO.NET Data Service 174](#_Toc234924957)

[24.3 Using the ADO.NET Data Services Protocol from the Client 177](#_Toc234924958)

[24.4 Summary 178](#_Toc234924959)

[25 Breaking Changes From May 2009 CTP 180](#_Toc234924960)

[25.1 Client Components 180](#_Toc234924961)

[25.1.1 DomainContext API Changes 180](#_Toc234924962)

[25.1.2 DomainDataSource API Changes 186](#_Toc234924963)

[25.1.3 Application Services API Changes 187](#_Toc234924964)

[25.2 Server Components 189](#_Toc234924965)

[25.2.1 DomainService Changes 189](#_Toc234924966)

[25.2.2 Delete Method Support 189](#_Toc234924967)

[25.2.3 Entity Framework and LINQ-to-SQL DomainServices 190](#_Toc234924968)

[25.2.4 SilverlightApplication and SeoSilverlightApplication Control Changes 191](#_Toc234924969)

[25.2.5 Removal of SharedAttribute Used in Code Generation 191](#_Toc234924970)

[25.3 Data Annotations 193](#_Toc234924971)

[25.3.1 Validator Class Changes 193](#_Toc234924972)

# Introduction

Microsoft Silverlight provides a strong foundation for building Rich Internet Applications (RIA). *Microsoft* *.NET RIA Services* introduced in this document further simplify Line of Business (LoB) RIA development. They complement the existing Data Access Layer and presentation components in the .NET framework and Silverlight. They build on the foundation of ASP.NET and codify, evolve and support some of the common patterns in web applications.

.NET RIA Services address the complexity of building N-tier applications through framework, tools and services. Framework components support prescriptive patterns for writing application logic and validation so that it can be easily used on the presentation tier. Tools add to existing Visual Studio capabilities by linking the client and mid-tier projects in a single solution and by enabling smart code generation in client projects. Accompanying services utilize the prescriptive pattern to support commonly used capabilities such as authentication and user settings management.

In this document, we will look at the key aspects of the .NET RIA Services preview. In this section, we will introduce the challenges of building an N-tier RIA and the approach taken to address the challenges. The following sections outline the core features as a set of task-focused quick starts. They complement the detailed walkthrough that introduces .NET RIA Services through a simple but detailed example.

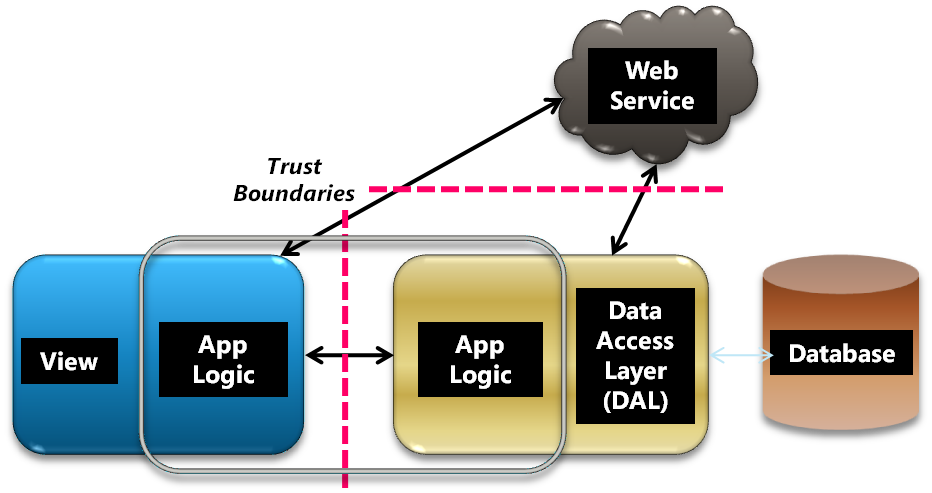
## Challenges of using data in N-tier applications

LoB applications use data that needs to flow across tiers. It may be created and used through basic Create, Read, Update and Delete (CRUD) operations or it may be accessed through domain-specific custom operations in encapsulated form such as an approval operation on an Expense Report. In an internet application, there is a trust boundary between the client and the mid-tier server. Hence, it is important to have a stylized way to express what resources are available to the client tier and what operations are permitted. Further, for productivity, it helps if the resources are easily available on the client and the operations can be easily invoked from a presentation tier with minimal additional plumbing.

In addition to defining the resources and operations, a developer needs to validate the data as it is created or updated. The validation needs to run on the presentation tier for giving immediate feedback to the user and on the mid-tier for ensuring that the rules defined for the application are enforced. Thus, there may be common validation across the tiers as well as tier-specific validations that need to be integrated with the resources and operations.

Finally, applications often need services that may be shared with other applications and too commonplace to be repeated. Authentication, roles, user settings are some of the services that are needed. They need to be managed and integrated with the usage of resources and operations in an N-tier application.

Figure 1 shows a simplified version of an N-tier application. .NET RIA Services focus on the box in between the view and the Data Access Layer (DAL) to address the multi-tier challenges outlined above.



Next, we will look at the key value propositions and core constructs used to address these challenges.

## Value propositions

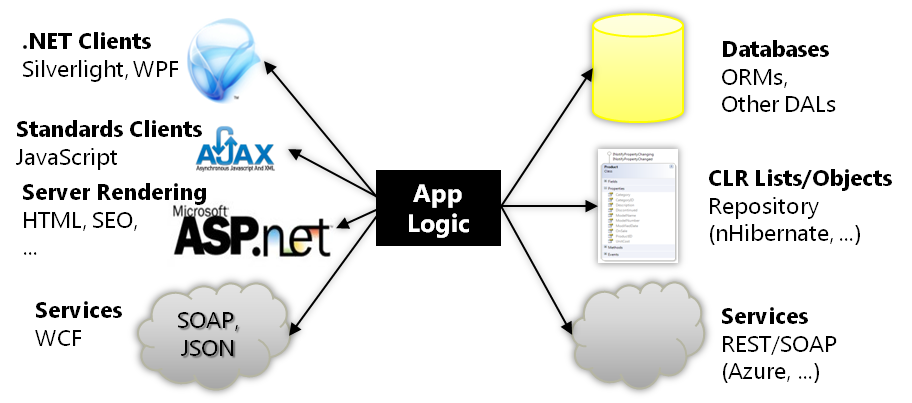
The key value propositions are:

1. Support for end-to-end use of data
2. Unified story for the development of the client and server parts of an application
3. Enhanced productivity through a prescriptive set of patterns and services targeting common scenarios

.NET RIA Services focus on the end-to-end use of data. It may be retrieved through a Data Access Layer (DAL) of your choice. It may be shaped for use in the presentation tier and annotated with suitable metadata for validation and access control. The new framework components support data and metadata flow across tiers through a controlled set of operations.

.NET RIA Services simplify application development across tiers and trust boundary by providing a set of tools to build the multiple tiers of an application together. The application logic remains aware of tier and trust boundaries but it leverages the power of .NET and Silverlight on respective tiers and utilizes the end-to-end flow of data and metadata described earlier. Code generation and shared code enhance the experience by providing multiple choices for sharing logic while respecting tier and trust boundaries.

.NET RIA Services define and support a pattern for exposing a set of operations on resources. A developer authors a DomainService to define a set of operations on resources. The framework and tools collaborate to generate code for a corresponding client-tier that can be used for data binding, validation etc. Services for authentication and user settings can be used out-of-the box with the operations and resources. The patterns and services utilize the foundation of the ASP.NET counterparts and extend some of the common ASP.NET web application patterns for use with a richer presentation tier enabled by Silverlight. The current preview illustrates the pattern primarily through the end-to-end experience with Silverlight and secondarily through an early glimpse of a basic html view. It includes direct support for Entity Framework and LINQ to SQL DALs as well as user-authored objects. However, it is designed to scale across different presentation technologies and DALs as shown in Figure 2.



## Key concepts

This section outlines some of the core concepts. They are covered in more details in the walkthrough document and in the following sections.

The main class for authoring operations is DomainService. The following example shows a DomainService class HRService for a Human Resources (HR) application. It derives from the .NET RIA Services base class for the LINQ to SQL DAL – LinqToSqlDomainService, which is parameterized by the LINQ to SQL DataContext type - AdventureWorksDataContext. It contains three operations: one for querying, one for updating and one for a custom operation to approve sabbatical.

The operations are defined on *entities* – i.e., objects with identity defined by keys. The Employee entity used in the DAL is also used by HRService. A developer may add optional metadata to shape the entity or to validate its properties.

[EnableClientAccess]

public class HRService :

LinqToSqlDomainService<AdventureWorksDataContext>

{

public IQueryable<Employee> GetEmployees()

{

// Restrict access as appropriate

return from e in Context.Employees

where e.SalariedFlag == true

select e;

}

public void UpdateEmployee(Employee currentEmployee)

{

// Check/enforce app-specific rules for changes

this.Context.Employees.Attach(currentEmployee,

this.ChangeSet.GetOriginal(currentEmployee));

}

public void ApproveSabbatical(Employee employee)

{

// Ensure that employee has required tenure

// and annual review ratings

}

}

.NET RIA Services takes note of the EnableClientAccess attribute applied to the DomainService and generates the corresponding client-tier classes: HRContext corresponding to HRService and a client version of the Employee entity. The classes are outlined below. The DomainContext class provides basic services such as invocation of operations, accumulation of results, object identity and change tracking in collaboration with the generated entity class. The EntityList provides a rich collection for data binding.

public sealed partial class HRContext : DomainContext

{

. . .

public EntityList<Employee> Employees { . . . }

// Creates an EntityQuery corresponding to the GetEmployees

// Query operation

public EntityQuery<Employee> GetEmployeesQuery() { . . . }

public void ApproveSabbatical(Employee emp) { . . . }

. . .

}

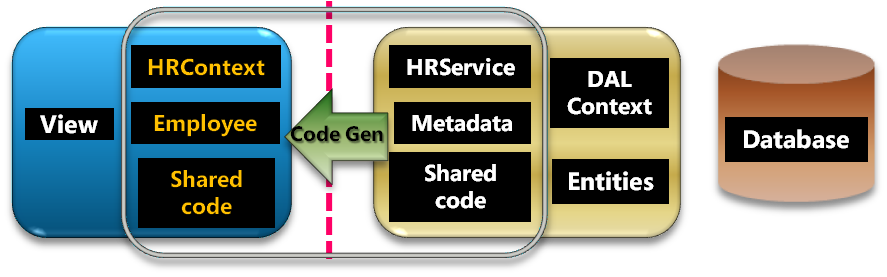
public sealed partial class Employee : Entity

{

…

}

In addition to the above, a developer may use shared code for validation or other purposes. Figure 3 shows the conceptual flow among the building blocks.



## Roadmap

A separate walkthrough document provides the best starting point for hands-on learning. This document contains additional quick starts that describe the concepts and outline how to complete specific, focused tasks that are complementary to the walkthrough. They complement the samples published elsewhere.

This overview covers the following:

1. Basic concepts and common tasks: Sections 2-15
2. Advanced concepts and specialized tasks: Sections 16-22
3. Using DomainService with ASP.NET and ADO.NET DataService: Sections 23-24
4. Breaking changes from May 2009 CTP: Section 25

## New in July 2009 CTP

Here is a list of what is new in the July 2009 Preview:

1. A new and improved model for handling operations on the client – you can more easily specify callback/event for an operation, track its status and cancel it.
2. Improved class library support
3. Extensibility points in generated code
4. Cleaner user model and better extensibility support for Application services
5. Better DomainDatasource capability for handling paging and updates.
6. Cleaner shared code support
7. First wave of ADO.NET Data Services alignment (add a DomainService to DataService for writing app logic / expose DomainService as DataService).
8. Bug fixes and initial performance improvement on the server

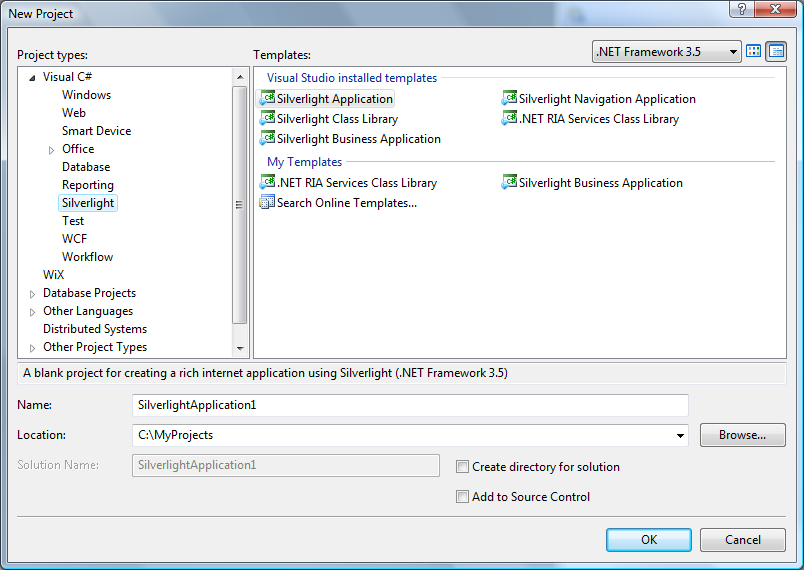
# Understanding N-tier Silverlight Application Projects

Microsoft .NET RIA Services simplifies the traditional N-tier application pattern by bringing together the ASP.NET and Silverlight platforms. Application logic is written on the mid-tier to control access to data via queries, updates, custom methods and service operations. The .NET RIA Services feature automatically generates equivalent proxy classes on the client tier to invoke these operations. And because both the mid-tier and client-tier are based on the same CLR (Common Language Runtime), the developer can write code that runs on both tiers.

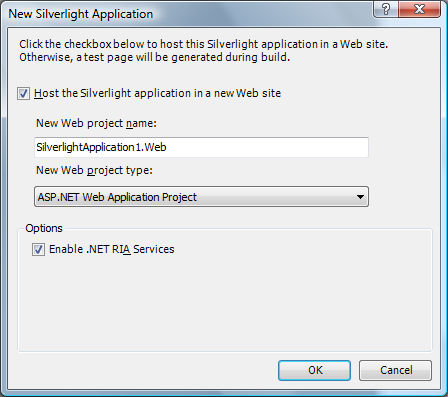
This document illustrates the key concepts by constructing a simple N-tier application within a single Visual Studio solution.

## Creating the project

Creating a .NET RIA Services N-tier application begins by creating a Silverlight web application project, using File | New Project[[1]](#footnote-1):

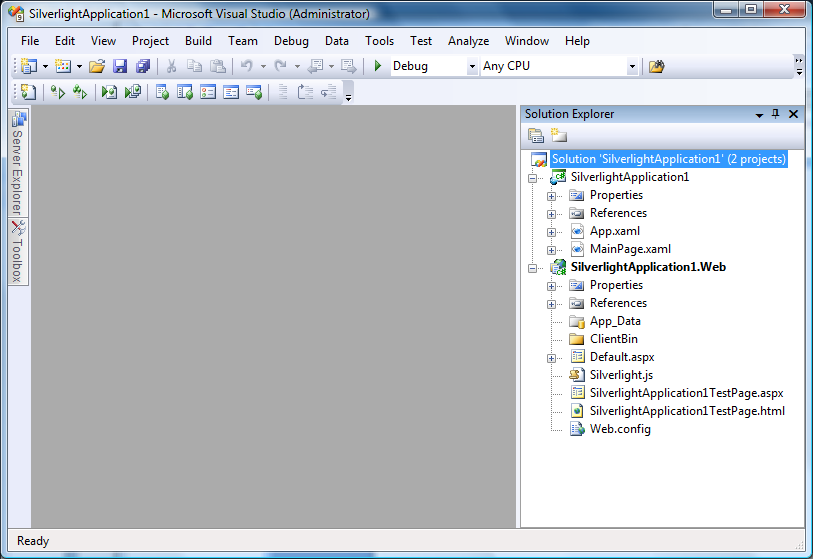


Proceeding from this dialog shows the New Silverlight Application dialog. If .NET RIA Services has been installed, this dialog displays the checkbox highlighted below:



By default this checkbox is *not* checked. To create an N-tier application using .NET RIA Services, you must manually check it.

The resulting solution looks like this:



At this point, we have created a single solution that consists of 2 projects:

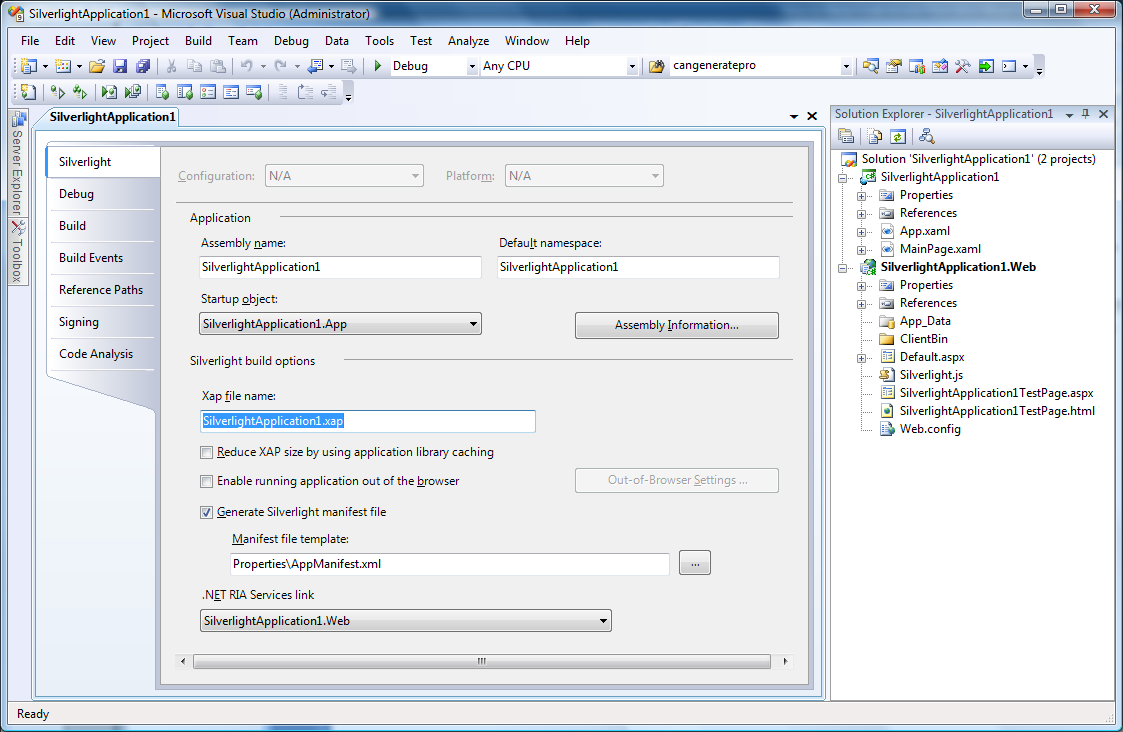
1. ***SilverlightApplication1*** – this project contains the Silverlight code. We refer to it as the *client* project. This will be our client tier.
2. ***SilverlightApplication1.Web*** – this project contains the ASP.NET web application code. We refer to it as the *server* project. This will be our mid-tier.

The main difference between this solution and a standard Silverlight web application is the “Enable .NET RIA Services” checkbox chosen above. This checkbox forms an association between the client and the server projects that permits application logic to be shared. It is known informally as the “RIA link” and is described in more detail in “Understanding Silverlight Client Code Generation” and “Understanding N-tier Class Libraries.”

## Converting an existing web application

As mentioned above, you can change your mind later whether the “RIA link” is enabled or disabled. This includes modifying existing web application projects you have already built. To change this setting, select the client project and examine its properties (right-click | Properties or use the menu item Project | SilverlightApplication1 Properties…). If .NET RIA Services has been installed, this screen will show the new drop-down box highlighted below.

To modify the “RIA link” between the Web and Silverlight application projects, choose a web application project from the available choices in the “.NET RIA Services Link” drop-down. To disable the link, choose “<No Server Project Set>.” In our example, this screen just confirms the choice we made earlier by checking the “Enable .NET RIA Services” checkbox.



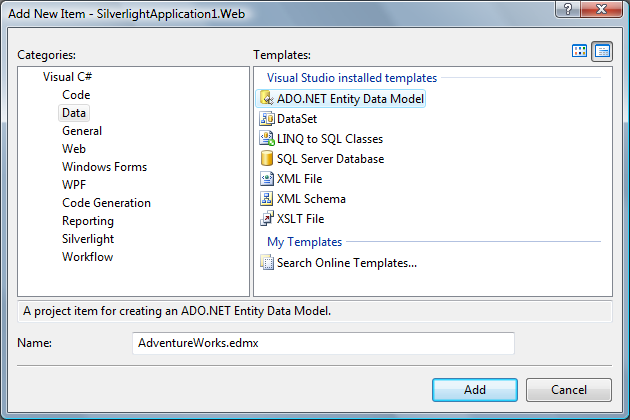
## Exposing data from the mid-tier

At this point, we have an ordinary Silverlight web application project that has enabled the new .NET RIA Services functionality. To demonstrate the new functionality, we will first make some data available to our mid-tier.

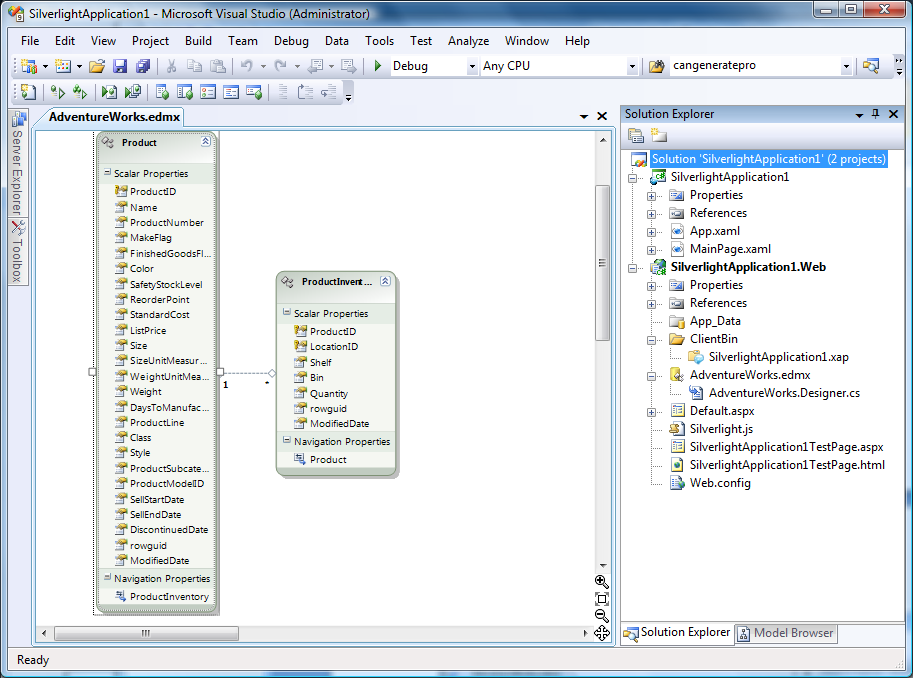
## Accessing data from the mid-tier

In this example, our mid-tier will use data from a database. The use of a database is not required by the .NET RIA Services framework. The client tier will be unaware of the data access technology or schema used by the mid-tier.

To use a database on our mid-tier, we will access the AdventureWorks database using the Entity Framework. We start by using Add New Item on the server project and selecting the Data category:



We’ll choose just 2 tables from the AdventureWorks database for this application:

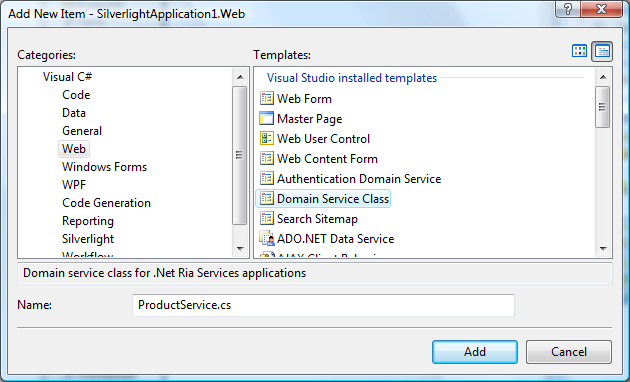


## Creating a DomainService on the mid-tier

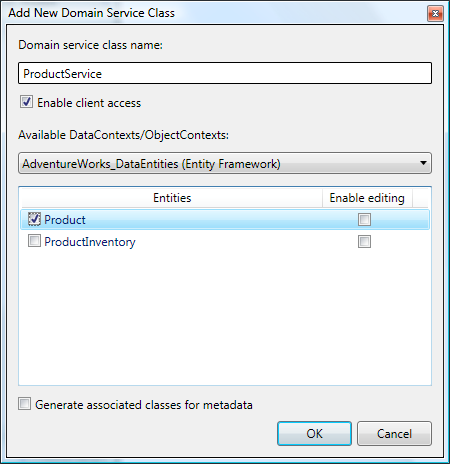
Now we come to the critical step that demonstrates the features of the .NET RIA Services framework; creating a *DomainService* on the mid-tier. A DomainService is a class that exposes entities and operations for a specific data domain. It is also where the developer adds application logic.

The .NET RIA Services framework contains the plumbing code to make the entities and operations declared in a DomainService class available to other tiers.

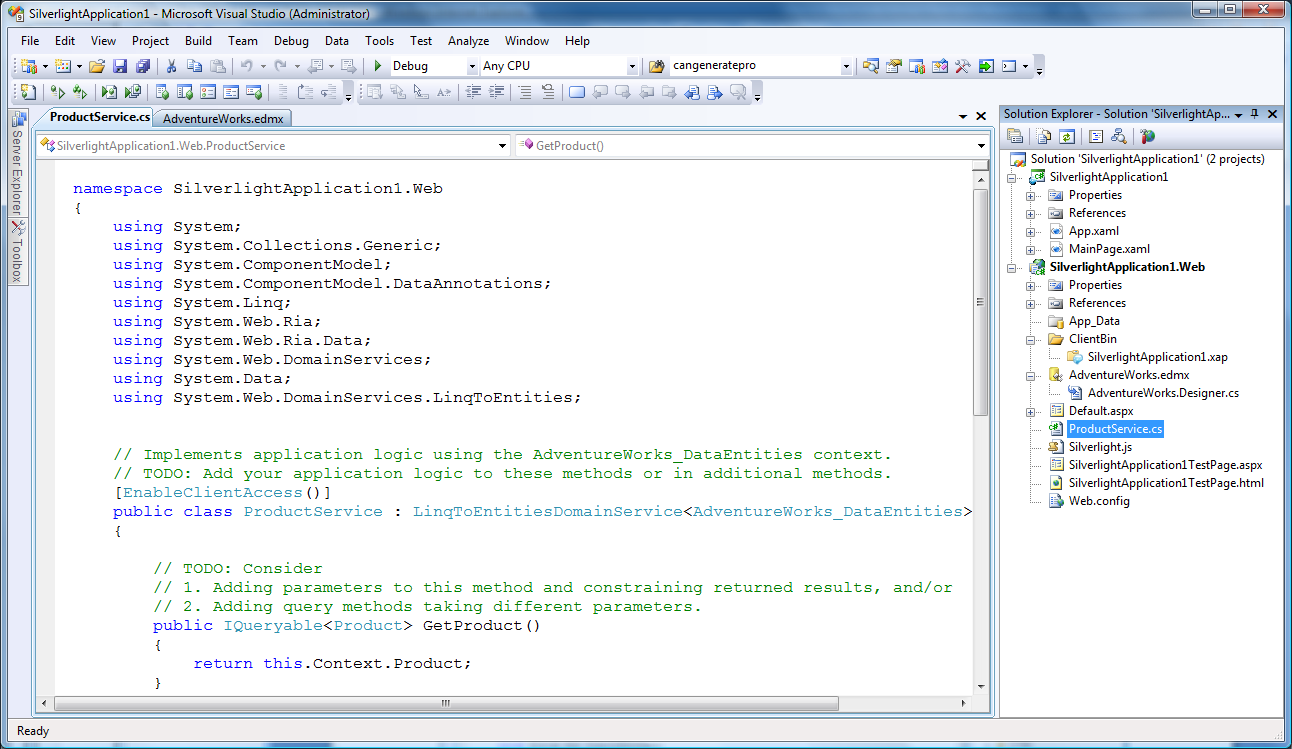
We will create our sample DomainService by selecting the server project and choosing Add New Item. Under the Web category is the new Domain Service Class template. We choose to call this new DomainService class ProductService:



This template invokes a wizard to help construct the DomainService class. In this example we’ll select the AdventureWorks model we just created, and we’ll expose only one entity; Product. We will not “Enable editing,” meaning the Product entity will be read-only[[2]](#footnote-2).



After we commit this dialog, the new DomainService class will be created in the selected project. In this case, we named it ProductService. Here is the generated code:



Notice the following characteristics of the generated code:

* The class derives from LinqToEntitiesDomainService, an abstract base class built into the .NET RIA Services framework[[3]](#footnote-3).
* The generic base class is bound to the AdventureWorks\_DataEntities class we created earlier
* The DomainService class is marked with [EnableClientAccess] to indicate it should be visible to the client tier
* A single GetProduct() query method has been generated because we asked to expose the Product entity
* Implementation code has been generated in GetProduct() to obtain data from the AdventureWorks\_DataEntities object context

This new DomainService class is where we would add our application logic. For example, we could modify the logic in the GetProduct() method to select only Products in stock. Or we might add a parameter to this method to specify a product category and alter our query to select only those products. We could also add new methods with parameters appropriate to other kinds of selection that make sense for our application.

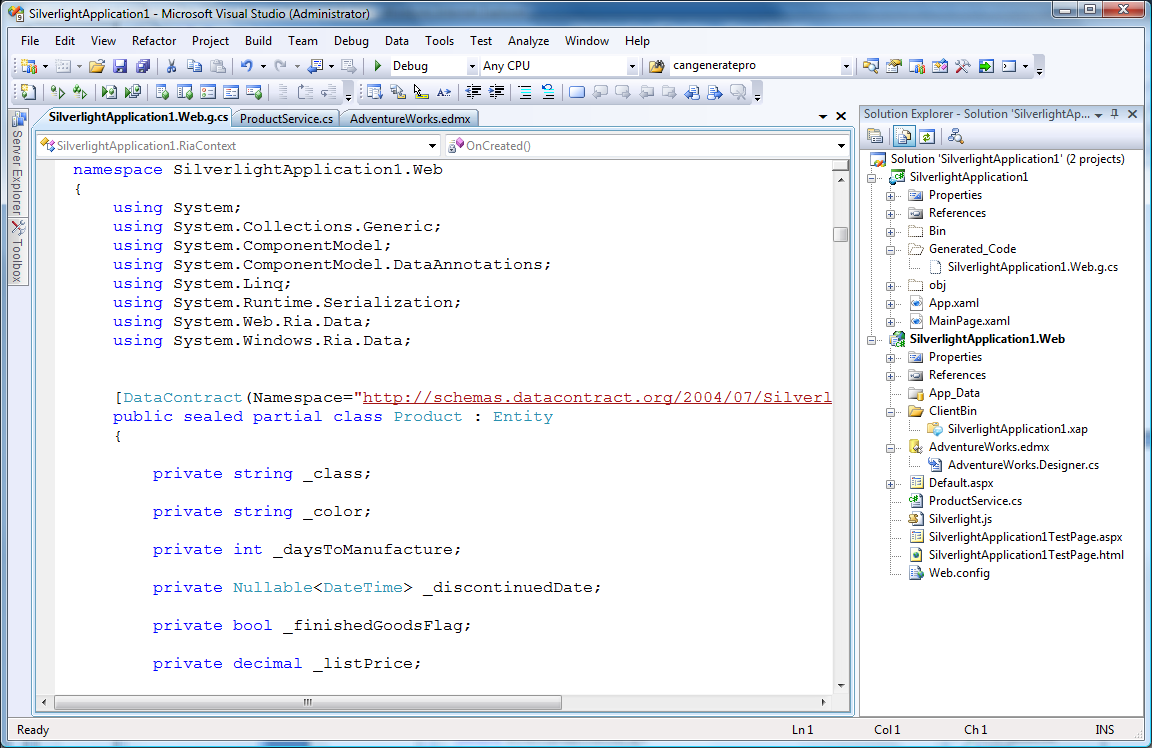
The important point here is that the DomainService class provides the public interface to the mid-tier data. The code generated by the wizard is just a starting point, and we will add our application logic here.

The second important point is the fact we can now run our application without any modifications, and the Product entities will be available on *both the mid-tier and the client tier*. The mechanisms that make this possible are described below.

## Accessing the data from the Silverlight client

Remember the “RIA link” we established when we created this application? Now it comes into play.

If we build the solution and click “Show All Files” in the client project, we will see this:



Notice that the client project now has a folder called Generated\_Code containing a generated file. The screenshot above shows a few lines of this file.

This generated file contains what we collectively call the *client proxy classes*. These classes have been synthesized from the DomainService class we created in the server project.

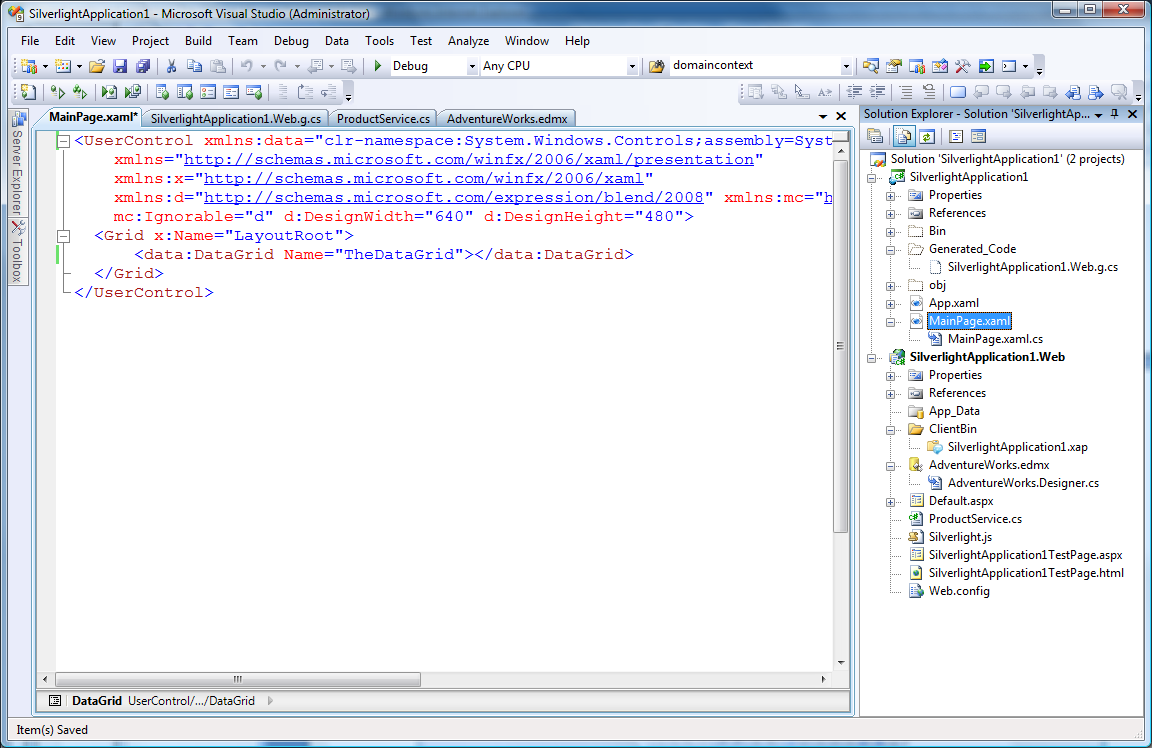
The mechanisms that generate these proxy classes and the manner in which they function are described in the “Understanding Silverlight Client Code Generation” section, but the important aspects to be aware of are:

* These client proxy classes are regenerated whenever the DomainService class on the server is modified and the client project or solution is rebuilt. You should not modify these proxy classes because they will be overwritten.
* A separate client proxy class is generated for every entity type exposed by each DomainService class – in this case the Product entity.
* A DomainContext class is generated to provide client-side access to the operations exposed by each DomainService. In this example a ProductContext class has been generated, and it exposes a GetProductQuery() method that can be used as an argument to the DomainContext.Load() method to invoke the GetProduct() query method in the ProductService DomainService class on the mid-tier. See “How to Query Entities.”

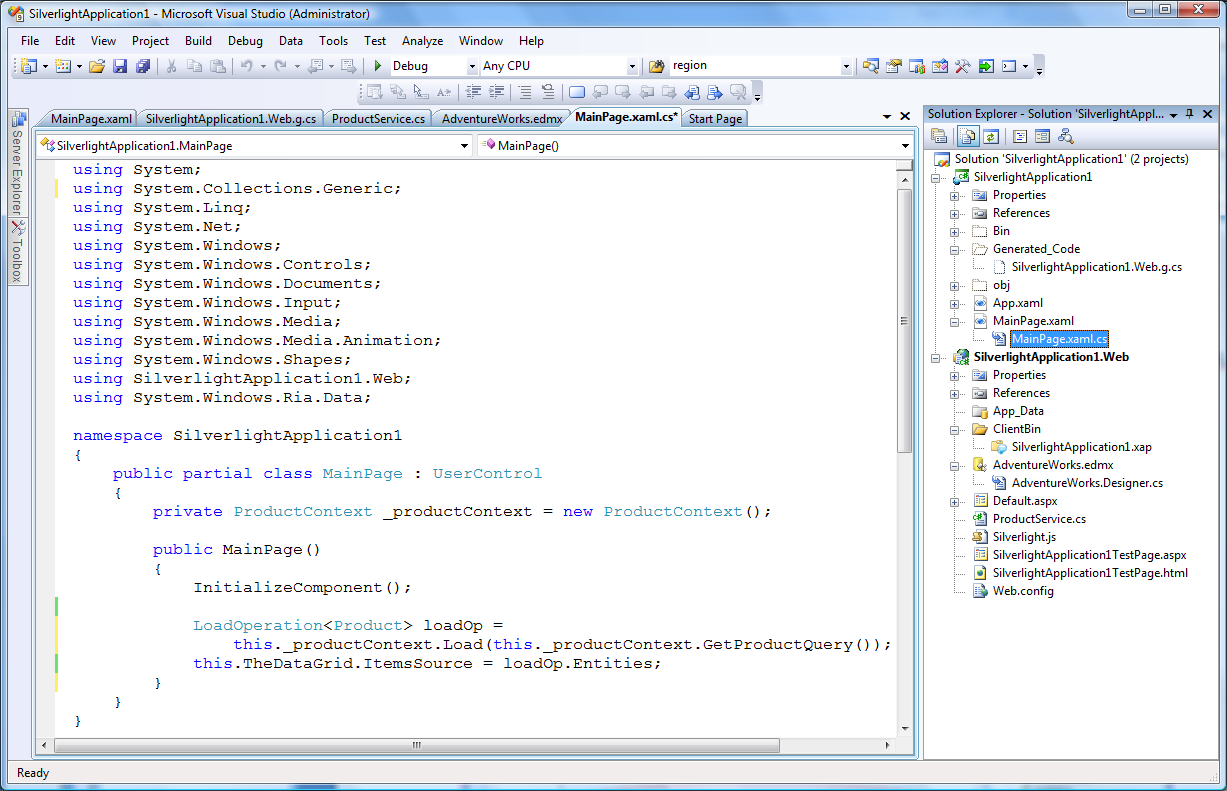
## Using the data on the Silverlight client

The generated file in the client project contains one *DomainContext* class for every DomainService on the server that was marked with the [EnableClientAccess] custom attribute. The Silverlight client can instantiate these DomainContexts and interact with them. The following example shows how one might do this to display the Products in a grid.

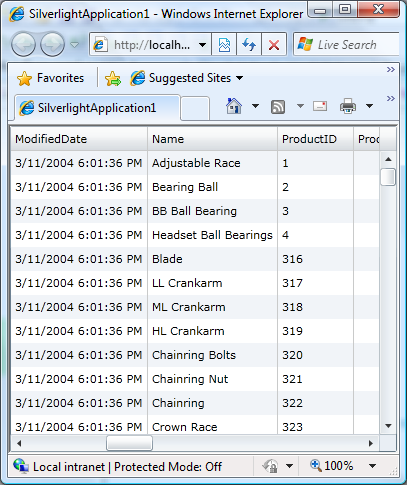
We’ll add a DataGrid to our client project like this:



And then we’ll add some code to the code-behind file to instantiate our generated DomainContext, retrieve our Products from the server, and data-bind them to the DataGrid:



We can now run this application and will see this:



## Summary

This section has provided a brief overview of the key concepts of an N-tier Silverlight application project using the .NET RIA Services framework. The key points to take from it are:

* A standard Silverlight web application project can be made into an N-tier application by establishing a “RIA link” between a Silverlight project and a Web project[[4]](#footnote-4).
* The simplest form of this N-tier application consists of 2 separate projects within a single solution; one for the ASP.NET server project (mid-tier), and the other for the Silverlight client project (client tier).
* A *DomainService* class in the server project exposes sets of entities and operations appropriate for a specific domain. Application developers place their custom application logic in this DomainService class.
* When the solution is built, client proxy classes are generated automatically into the Silverlight client project[[5]](#footnote-5), corresponding to the entities and operations exposed by the server’s DomainService.
* The Silverlight client code interacts with the generated *DomainContext* proxy class to obtain and manipulate the corresponding entities.
* Both the server and client tiers operate on a common set of entity types, making it possible to share business logic and validation rules across the tiers.

# Understanding Silverlight Client Code Generation

The Microsoft .NET RIA Services framework makes it easy to build an N-tier application consisting of an ASP.NET mid-tier and one or more Silverlight client tiers. This framework allows a developer to expose domain-specific entities and operations on the mid-tier that can also be used by the Silverlight client tiers.

The entities and operations exposed by the mid-tier are made available to the Silverlight client tier through automatic generation of *client proxy classes* into the Silverlight client project.

This section describes how the code generation mechanisms work.

## The sample application

We’ll start with a small application to illustrate the mechanisms involved. And to help strip away the layers of magic, we’ll keep it as simple as possible. Here is our application:



We created this application using the normal “File | New Project | Silverlight Application” template described in “Understanding N-tier Silverlight Application Projects”. We then customized the web application to use a small custom-built data store. Let’s look at the files we created more closely.

### Sample City entity and store

For this example, we’ve chosen to expose a simple POCO (Plain Old CLR Object) entity called City. Here is the City.cs class:

using System;

using System.ComponentModel.DataAnnotations;

namespace SilverlightApplication1.Web.Cities

{

public partial class City

{

[Key]

public string Name {get;set;}

[Key]

public string State { get; set; }

}

}

And we’ve chosen to make a very simple in-memory data store called CityData.cs:

using System;

using System.Collections.Generic;

namespace SilverlightApplication1.Web.Cities

{

public class CityData

{

private List<City> \_cities = new List<City>( new City[] {

new City() { Name="Redmond", State="WA" },

new City() { Name="Bellevue", State="WA" }

});

public IList<City> Cities

{

get

{

return this.\_cities;

}

}

}

}

Obviously, we would use a database for our real application, but it’s easier to understand the code-generation process if we remove all unnecessary moving parts.

### Sample DomainService

The ASP.NET server project provides access to domain-specific entities and operations through a DomainService class. Here is the DomainService class we’ve created by hand to provide read-only access to our City entities:

namespace SilverlightApplication1.Web.Cities

{

using System;

using System.Collections.Generic;

using System.Web.Ria;

using System.Web.DomainServices;

[EnableClientAccess()]

public class CityService : DomainService

{

private CityData \_cityData = new CityData();

public IEnumerable<City> GetCities()

{

return this.\_cityData.Cities;

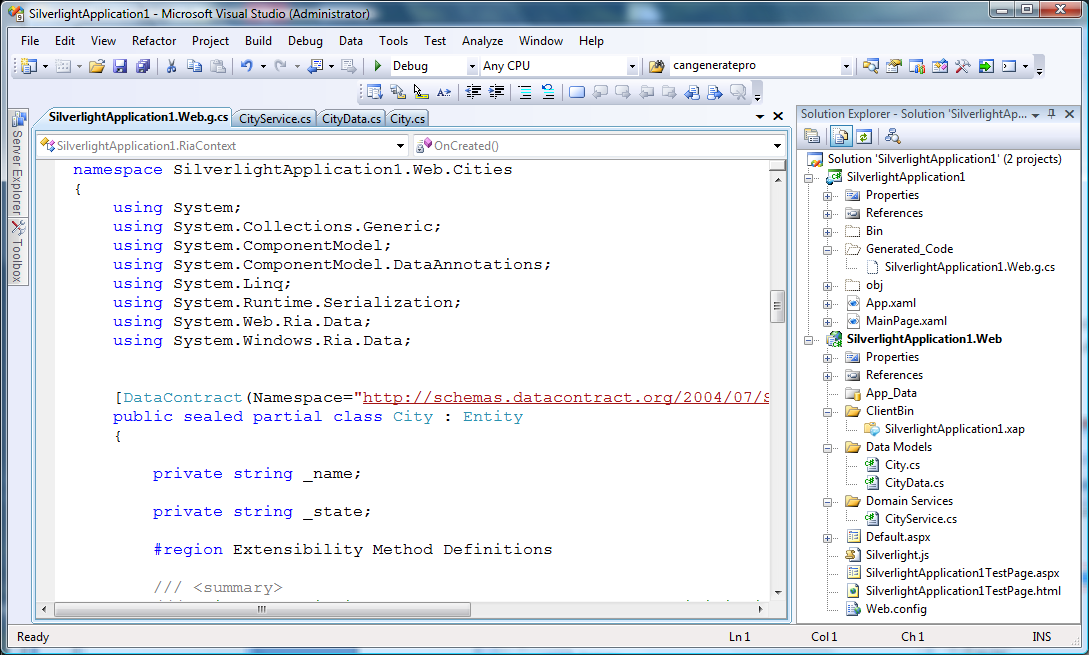
}

}

}

### The generated code

We now build the solution and notice that a file has been generated in the Silverlight client project. To see this file, you must select the SilverlightApplication1 client project and click the “Show All Files” icon at the top of the Solution Explorer:



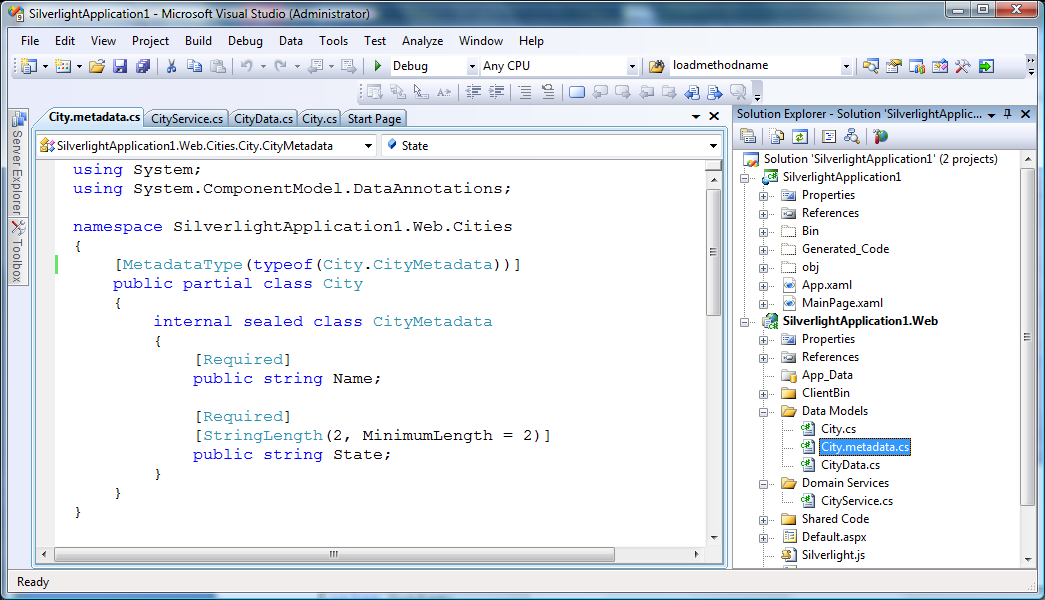
The complete generated code is shown in [Generated Code](#_Appendix_A_–). You will notice two important things:

* The generated code defines a City class that matches the shape and namespace of the City class on the server.
* The generated code defines a CityContext class that exposes a GetCitiesQuery() method. By passing this method’s return value to the DomainContext’s Load() method, we can retrieve City entities from the CityService DomainService we defined on the server. For more details see “How to query entities”.

### Adding metadata to the entity

We could run the application at this point, and the Silverlight application could start using the CityContext and City entity types. But we want to demonstrate two more aspects of code generation; metadata classes and shared files.

We want to add some validation rules that are applied on both tiers, so we’ll create a new metadata class (affectionately called the “buddy class”). Here is the class we wrote to do this:



The name of this file or class is not important, but the naming pattern shown here enhances comprehension.

Here is the code for the City.metadata.cs class above:

using System;

using System.ComponentModel.DataAnnotations;

namespace SilverlightApplication1.Web.Cities

{

[MetadataType(typeof(City.CityMetadata))]

public partial class City

{

internal sealed class CityMetadata

{

[Required]

public string Name;

[Required]

[StringLength(2, MinimumLength = 2)]

public string State;

}

}

}

Metadata classes provide a convenient way to attach metadata to an entity without actually modifying the corresponding members on the entity itself. This is described in “How to use metadata classes”.

The rules for using a metadata class are:

1. The entity class must be marked with MetadataType attribute to specify its associated metadata class
2. The metadata class must contain only members whose name match members in the entity class

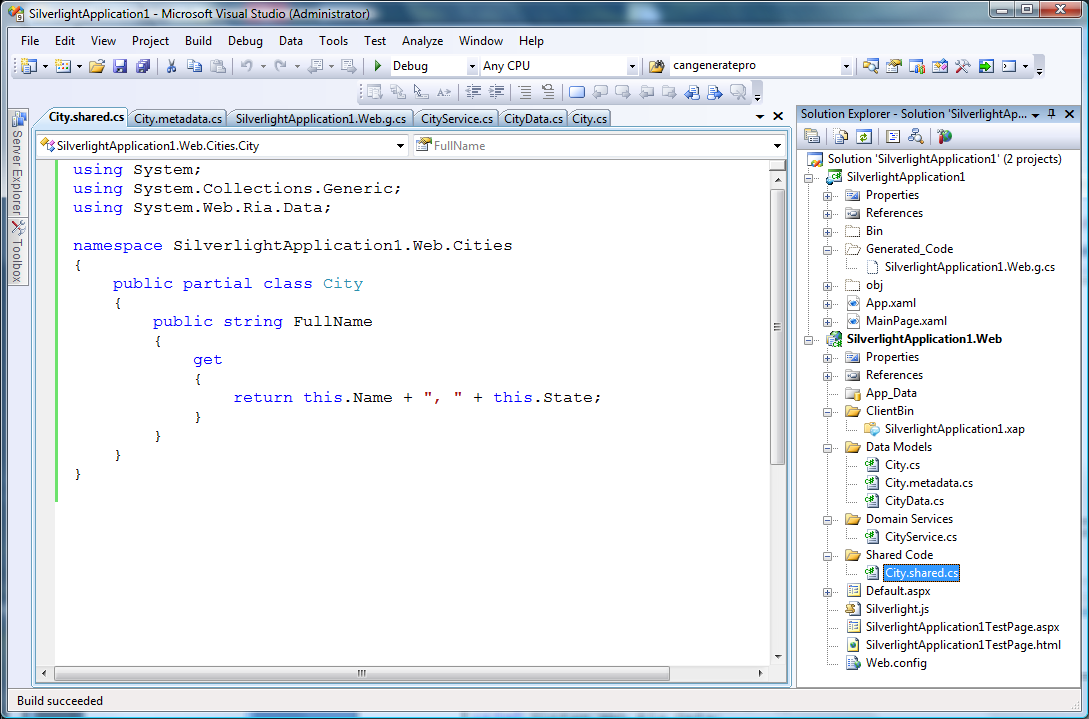
The metadata class may omit members it does not want to extend, but it may not contain members not in the entity class. The members are required only to match in name, which explains why Name is a property in the City class but a field in the metadata class. Metadata members are not required to be of the same type as their corresponding entity member, but it helps readability if they are.

For every member of the entity class that has a matching member in the metadata class, the custom attributes are combined. The metadata class exists only on the server, and when code-generation creates the client proxy class for the entity, the custom attributes are combined into the generated class.

### Adding shared code

We want to illustrate one more feature handled by the client code generation process; files may be shared between the server and client projects. This is described in more detail in “How to Share Code Across Tiers”.

We start by creating City.shared.cs. This naming pattern is important, because it is this name that causes the code-generation task to recognize the file as shared[[6]](#footnote-6). The location of the file is not important, but it must be included in the server’s list of files. We chose to put it in a folder called “Shared Code” just to keep things organized. Here is the solution with this file:



And here is the code we wrote for the City.shared.cs file:

using System;

namespace SilverlightApplication1.Web.Cities

{

public partial class City

{

public string FullName

{

get

{

return this.Name + ", " + this.State;

}

}

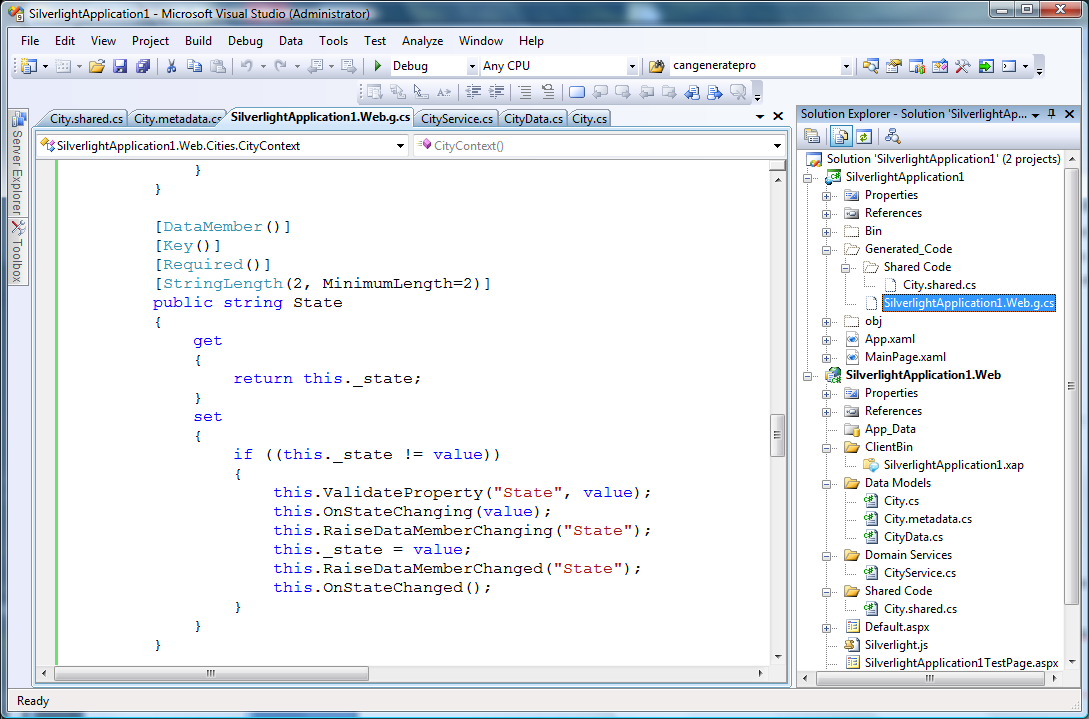
}

}

We have effectively created a computed property on the City entity. This property will be visible on both the server and client tiers. By approaching it this way, we have not altered the actual entity structure in our in-memory store, nor have we increased the size of the payload when moving entities across the wire. But from the standpoint of application logic on either tier, the City entity just gained a new property.

### The final generated code

If we build the solution again, we will see the following additions to the client project:



Notice these additions:

* The validation attributes we attached to the City entity via the metadata class on the mid-tier have been propagated into the generated code on the Silverlight client tier.
* The shared file we created on the mid-tier has been propagated verbatim into our client project, including its folder structure.

## How it works

This section explains how the code generation mechanisms work. Code generation will occur only if the following 2 conditions are met:

1. The .NET RIA Services framework has been installed, and
2. The Silverlight client project has a “RIA Link” to the server project

If these preconditions are met, a build of either the solution or the Silverlight client project will cause code-generation. An explicit build dependency injected into the solution forces the server project to build before code-generation.

The “RIA Link” is described in “Understanding N-tier Silverlight Application Projects”.

### The code-generation algorithms

Code generation begins by examining all assembly references in scope of the server project. Pseudo-code for this process looks like this:

Foreach (Assembly either built or referenced)

Foreach (DomainService marked with [EnableClientAccess])

Generate a DomainContext class on the Silverlight client

Foreach (Entity exposed by DomainService)

Generate entity proxy class on the Silverlight client

In other words:

* All assemblies referenced or built by the server project are analyzed.
* All classes derived from DomainService that are marked with the [EnableClientAccess] custom attribute are analyzed
* All public methods of the DomainService are analyzed to determine which entity types it exposes (via method return types or method input parameters)

If no public DomainServices with the [EnableClientAccess] custom attribute are found in any of the server project’s assemblies, no generated file will be produced.

### Generated entity proxy classes

Each entity class exposed or consumed by the server’s DomainService will cause a corresponding entity proxy class to be generated, with these characteristics:

* The entity proxy class will use the same namespace as the server’s entity class
* The entity proxy class name will match the server’s entity class name
* Every public property on the server’s entity class name will be exposed by the entity proxy class[[7]](#footnote-7)
* Code will be generated for the proxy class property setters to:
  + Raise INotifyPropertyChanged notifications, allowing them to be data-bindable
  + Perform validation
* Custom attributes attached to the entity class or its properties will be propagated into the proxy class if possible (see below).

### Generated custom attributes

Custom attributes attached to the entity class or any of its properties can provide useful metadata to the client tier, including such things as validation, UI hints, an indication of [Key] fields, etc. The code generator will therefore propagate an entity’s custom attributes into the proxy class, subject to these rules:

* The custom attribute type itself must be available on the client project
* Types specified in the attribute declaration must be available on the client project
* The custom attribute type must be structured in such a way that it:
  + exposes public setters for all its properties, or
  + exposes constructors that allow properties without setters to be specified

In short, the code generator will propagate custom attributes only if doing so will not cause compilation errors in the client project. In this way, DAL-specific custom attributes that exist only in assemblies available on the server are not propagated to the proxy classes.

However, if you examine the generated code and find that some custom attributes you wanted were not propagated to the proxy classes, *check your assembly references for the client project*. Add whatever assembly references you need to make the attribute types available on the client, and then rebuild the project.

### The metadata class

When creating the client proxy classes for the entities, the code-generator will automatically merge[[8]](#footnote-8) validation attributes it finds in an associated metadata class, if one is present.

No metadata class will appear on the client. Instead, the generated proxy class for the entity will reflect the results of merging the custom attributes during the code generation process.

For more details, refer to “How to use metadata classes”.

### The DomainContext class

In addition to one proxy class per entity, the generated file on the Silverlight client also contains a DomainContext class. Look at the generated code in [Generated Code](#_Appendix_A_–) to see what was generated for our sample application. CityContext is the generated DomainContext.

The code-generation task examines every public method in the server’s DomainService class and generates the DomainContext class according to these rules:

* The DomainContext class uses the same namespace as its DomainService class
* Three constructors are created:
  + A default constructor that embeds the URI necessary to communicate with the DomainService over http using a built-in DomainClient class.
  + A constructor that permits the client to specify an alternate URI
  + A constructor that permits the client to provide a custom DomainClient implementation (typically for unit testing or redirection to a custom transport layer)
* Every query method in the DomainService generates a corresponding EntityQuery method in the DomainClient[[9]](#footnote-9). For examples, the GetCities() method in the DomainService generates GetCitiesQuery() in the CityContext class. The value returned from this method can be passed to the DomainContext.Load() method to invoke the GetCities() query on the mid-tier.
* Custom Methods and Service Operations defined in the DomainService cause corresponding methods to be generated in the entity and DomainContext proxy classes.
* Public DomainService methods that perform inserts, updates, or deletes cause the generated EntityLists in the DomainContext to be constructed with flags indicating which of these operations are permitted on the client.

For more details on how the DomainContext works, refer to these sections:

“How to query entities”

“How to modify entities”

“Using Custom Methods and Service Operations”

### The shared file copy algorithms

During the code-generation phase, all files named according to the pattern \*.shared.cs or \*.shared.vb will be copied verbatim into the client project. Any folder structure in the server project will be replicated in the client. The code generation process does not interpret these files; they are simply copied.

### The automatic detection of shared code

To avoid generating duplication definitions in the proxy classes, the code generator must detect which types and members already exist in the client project.

In our example, we used City.shared.cs to add the FullName property to the City entity. Due to the name of this file, it was copied verbatim into the client project. In other words, even without proxy class code generation, this property was pre-defined in the City class on both tiers. If the code generator were to generate a property getter/setter pair in the entity proxy class for this property, it would create a compilation error.

Therefore, when the code generator creates the client proxy classes and is generating the property getter/setter pairs, it will automatically determine which types and members already exist on the client to avoid generating duplicates.

This is described in more detail in “How to Share Code Across Tiers”.

## Summary

The Microsoft .NET RIA Services framework simplifies the job of creating an N-tier application that combines the ASP.NET and Silverlight platforms.

The application developer exposes a set of domain-specific entities and operations using a DomainService class, and the framework automatically generates corresponding proxy classes into the Silverlight client project. The developer writes code in the Silverlight client project to interact with the generated DomainContext class to use these generated entity proxy classes and to invoke operations exposed by the DomainService.

The .NET RIA Services framework handles the plumbing to move entities and to invoke operations across the wire. The framework also handles low-level details like change-tracking and caching.

In this way, the application developer can focus on the core application logic using a common set of entity classes, validation rules and operations than span the tiers.

## Generated code

Here is a complete copy of the code generated in the client project of our sample application.

//------------------------------------------------------------------------------

// <auto-generated>

// This code was generated by a tool.

// Runtime Version:2.0.50727.3053

//

// Changes to this file may cause incorrect behavior and will be lost if

// the code is regenerated.

// </auto-generated>

//------------------------------------------------------------------------------

namespace SilverlightApplication1

{

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Web.Ria.Data;

using System.Windows.Ria.Data;

/// <summary>

/// Context for the RIA application.

/// </summary>

/// <remarks>

/// This context extends the base to make application services and types available

/// for consumption from code and xaml.

/// </remarks>

public sealed partial class RiaContext : System.Windows.Ria.RiaContextBase

{

#region Extensibility Method Definitions

/// <summary>

/// This method is invoked from the constructor once initialization is complete and

/// can be used for further object setup.

/// </summary>

partial void OnCreated();

#endregion

/// <summary>

/// Initializes a new instance of the RiaContext class.

/// </summary>

public RiaContext()

{

this.OnCreated();

}

/// <summary>

/// Gets the context that is registered as a lifetime object with the current application.

/// </summary>

/// <exception cref="InvalidOperationException"> is thrown if there is no current application,

/// no contexts have been added, or more than one context has been added.

/// </exception>

/// <seealso cref="Application.ApplicationLifetimeObjects"/>

public new static RiaContext Current

{

get

{

return ((RiaContext)(System.Windows.Ria.RiaContextBase.Current));

}

}

}

}

namespace SilverlightApplication1.Web.Cities

{

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.ComponentModel.DataAnnotations;

using System.Linq;

using System.Runtime.Serialization;

using System.Web.Ria.Data;

using System.Windows.Ria.Data;

[DataContract(Namespace="http://schemas.datacontract.org/2004/07/SilverlightApplication1.Web.Cities")]

public sealed partial class City : Entity

{

private string \_name;

private string \_state;

#region Extensibility Method Definitions

/// <summary>

/// This method is invoked from the constructor once initialization is complete and

/// can be used for further object setup.

/// </summary>

partial void OnCreated();

partial void OnNameChanging(string value);

partial void OnNameChanged();

partial void OnStateChanging(string value);

partial void OnStateChanged();

#endregion

/// <summary>

/// Default constructor.

/// </summary>

public City()

{

this.OnCreated();

}

[DataMember()]

[Key()]

[Required()]

public string Name

{

get

{

return this.\_name;

}

set

{

if ((this.\_name != value))

{

this.ValidateProperty("Name", value);

this.OnNameChanging(value);

this.RaiseDataMemberChanging("Name");

this.\_name = value;

this.RaiseDataMemberChanged("Name");

this.OnNameChanged();

}

}

}

[DataMember()]

[Key()]

[Required()]

[StringLength(2, MinimumLength=2)]

public string State

{

get

{

return this.\_state;

}

set

{

if ((this.\_state != value))

{

this.ValidateProperty("State", value);

this.OnStateChanging(value);

this.RaiseDataMemberChanging("State");

this.\_state = value;

this.RaiseDataMemberChanged("State");

this.OnStateChanged();

}

}

}

public override object GetIdentity()

{

return EntityKey.Create(this.\_name, this.\_state);

}

}

public sealed partial class CityContext : DomainContext

{

#region Extensibility Method Definitions

/// <summary>

/// This method is invoked from the constructor once initialization is complete and

/// can be used for further object setup.

/// </summary>

partial void OnCreated();

#endregion

/// <summary>

/// Default constructor.

/// </summary>

public CityContext() :

this(new HttpDomainClient(new Uri("DataService.axd/SilverlightApplication1-Web-Cities-CityService/", System.UriKind.Relative)))

{

}

/// <summary>

/// Constructor used to specify a data service URI.

/// </summary>

/// <param name="serviceUri">

/// The CityService data service URI.

/// </param>

public CityContext(Uri serviceUri) :

this(new HttpDomainClient(serviceUri))

{

}

/// <summary>

/// Constructor used to specify a DomainClient instance.

/// </summary>

/// <param name="domainClient">

/// The DomainClient instance the DomainContext should use.

/// </param>

public CityContext(DomainClient domainClient) :

base(domainClient)

{

this.OnCreated();

}

public EntityList<City> Cities

{

get

{

return base.Entities.GetEntityList<City>();

}

}

/// <summary>

/// Returns an EntityQuery for query operation 'GetCities'.

/// </summary>

public EntityQuery<City> GetCitiesQuery()

{

return base.CreateQuery<City>("GetCities", null, false, true);

}

protected override EntityContainer CreateEntityContainer()

{

return new CityContextEntityContainer();

}

internal sealed class CityContextEntityContainer : EntityContainer

{

public CityContextEntityContainer()

{

this.CreateEntityList<City>(EntityListOperations.None);

}

}

}

}

# How to Query Entities

This section explores the details of the **R**ead operation in the .NET RIA Services CRUD (**C**reate, **R**ead, **U**pdate, **and Delete**) model, which corresponds to the retrieval of data from underlying data store(s). Readers are assumed to be familiar with the concept of DomainService, through which domain operations for data manipulations are defined.

## Query Methods

This section describes the query support in .NET RIA Services in more details. We will cover how query methods are defined, how data can be retrieved on the client, and how to filter data. We will also cover the use of query events for synchronizations.

## Defining Query Methods

To enable a specific Entity type to be retrievable by the client, one needs to define a corresponding query method in the domain service.

The query method signature is defined as follows:

1. It returns a single instance T, or an IEnumerable<T> or IQueryable<T> where T is a valid Entity type. To be an entity, a Type must have a key member, and must be a public, non-generic concrete type.
2. It takes 0 or more parameters (which are restricted to a set of supported types described later in this section).
3. It is adorned with QueryAttribute. If you’re using naming convention (see separate section “Understanding Convention and Configuration” for more on convention-based approach), a method is inferred as a query method if it meets the above criteria and doesn’t match any of the naming conventions for other operation types.

As an example, the following are two ways one can define query methods for the Employees table in the AdventureWorks domain service.

[EnableClientAccess()]

public class AdventureWorksDomainService :

LinqToEntitiesDomainService<AdventureWorksEntities>

{

// convention based collection returning

public IQueryable<Employee> GetEmployees()

{

return this.Context.Employee;

}

// convention based singleton returning

public Employee GetEmployeeById(int empId)

{

return this.Context.Employees.Single(p => p.EmployeeID == empId);

}

}

[EnableClientAccess()]

public class AdventureWorksDomainService :

LinqToEntitiesDomainService<AdventureWorksEntities>

{

// attribute based collection returning

[Query]

public IQueryable<Employee> ReturnAllEmployees()

{

return this.Context.Employee;

}

// attribute based singleton returning

[Query(IsComposable=false)]

public Employee GetEmployeeById(int empId)

{

return this.Context.Employees.Single(p => p.EmployeeID == empId);

}

}

Note that when using the QueryAttribute for singleton returning query operations, IsComposable must be set to ‘false’. The above example shows a query parameter in the context of a singleton returning query method, and the following example shows how to retrieve all employees who match the input marital status.

[EnableClientAccess()]

public class AdventureWorksDomainService :

LinqToEntitiesDomainService<AdventureWorksEntities>

{

// get employees with the specified marital status

public IQueryable<Employee> GetEmployees(string maritalStatus)

{

return this.Context.Employee.Where(

e => e.MaritalStatus == maritalStatus);

}

}

It is important to note that domain operations do not support method overloads. More concretely, if you want to have multiple query methods in the same domain service with different number of input parameters (e.g. GetEmployees() and GetEmployees(string)), you’ll need to name them differently.

## Client Code Generation for Query Methods

When adorning a domain service with EnableClientAccessAttribute, the .NET RIA Services framework automatically generates method proxies on the client. This allows a Silverlight application to call a domain operation on the client that results in the corresponding operation invocation on the middle tier. All query operations are performed in an asynchronous manner, meaning that the application thread will not be blocked during the processing of the query results, allowing it to respond to other events, like UI changes, etc.

When a query method is defined in the domain service, the code generator automatically creates the following:

* A **query factory** method that returns a strongly typed EntityQuery<T> instance corresponding to the mid-tier query method. For the GetEmployees method above the corresponding query factory method would be GetEmployeesQuery(int maritalStatus). The generated query factory method takes the same parameters that the mid-tier query operation expects. This query factory method will have the same name as the query operation, with “Query” appended to it.
* The **entity Type** of the query method, along with its serializable members[[10]](#footnote-10)

### Using Generated Query Methods

The usage pattern for client query is to create an EntityQuery instance, apply LINQ query operators to it as needed, and pass the query instance into DomainContext.Load. For example:

EntityQuery<Employee> query =

from e in \_ctxt.GetEmployeesQuery('M')

where e.BirthDate < new DateTime(1974, 8, 13)

select e;

LoadOperation<Employee> lo = \_ctxt.Load(query);

For details on LoadOperation and the functionality it provides, see the section “Asynchronous Domain Operations”.

There are four\* overloads of the DomainContext.Load method:

* An overload taking an EntityQuery instance to load
* An overload taking an EntityQuery instance and a MergeOption parameter
* An overload taking an EntityQuery instance, a callback (with signature Action<LoadOperation<T>>) and optional userState that will be passed through to the callback via LoadOperation.UserState
* An overload taking an EntityQuery, a MergeOption, a callback, and optional userState

\**There is actually a fifth non-generic overload that is marked EditorBrowsable(Never) attribute. This method is virtual and is called by all the other overloads. This can be used to inject before/after logic for all query operations. In addition, it can also be used by non-strongly typed framework code that can’t use the generic APIs.*

For the above GetEmployee example, the generated query factory method is:

public sealed partial class AdventureWorksDomainContext : DomainContext

{

. . .

/// <summary>

/// Returns an EntityQuery for query operation 'GetEmployees'.

/// </summary>

public EntityQuery<Employee> GetEmployeesQuery(char maritalStatus)

{

Dictionary<string, object> parameters =

new Dictionary<string, object>();

parameters.Add("maritalStatus", maritalStatus);

return base.CreateQuery<Employee>(

"GetEmployees", parameters, false, true);

}

}

## Using the IncludeAttribute

The IncludeAttribute can be applied to association members to shape the generated client entities as well as to influence serialization. Below we explore two uses of this attribute.

### Returning Related Entities

When querying entity Types with associations to other entities, often you’ll want to return the associated entities along with the top level entities returned by the query. For example, assume your DomainService exposes Products, and for each Product returned you also want to return its associated ProductSubCategory. The first thing to do is ensure that the associated entities are actually returned from the data source when Products are queried. The mechanism used to load the associated entities is a DAL specific thing. In this LINQ to SQL example, the code would be:

public IQueryable<Product> GetProducts()

{

DataLoadOptions loadOpts = new DataLoadOptions();

loadOpts.LoadWith<Product>(p => p.ProductSubcategory);

this.Context.LoadOptions = loadOpts;

return this.Context.Products;

}

With this code, all ProductSubCategories are returned from the DAL when Products are queried. The next step is to indicate to .NET RIA Services that a ProductSubCategory entity should be code generated, as should the bi-directional association between the two Types. To do this you need to apply the **IncludeAttribute** to the association member.

IncludeAttribute has both **serialization semantics** and **data-shape semantics**. For both serialization of results to the client as well as the client object model we code-gen, .NET RIA Services will only ever expose data and types from the service that you have explicitly exposed from your service.

* If a Type is exposed via a query method from the service a corresponding proxy class will be generated on the client for that Type. In addition any associations referencing that Type from other exposed types will also be generated.
* If a Type is not exposed by a query method, but there is an association on another exposed Type that is marked with IncludeAttribute, a corresponding proxy class will be generated on the client for that Type.

Those points define what types are defined on the client and their shapes. In addition to these data shape semantics, IncludeAttribute also has serialization semantics : during serialization, only associations marked with IncludeAttribute will actually be traversed. All of this ensures that only information you've explicitly decided to expose is exposed.

Therefore, we need to apply the IncludeAttribute to the Product.Category association member. We do this by adding the attribute to the buddy metadata class:

internal sealed class ProductMetadata

{

[Include]

public ProductSubcategory ProductSubcategory;

}

With this in place we’ll get a ProductSubCategory entity generated on the client, as well as the Product.ProductSubCategory association member. This works the same way for collection associations. For example, to include all Products when querying a ProductSubCategory, you’d put the IncludeAttribute on the ProductSubCategory.Products member.

### Denormalizing Associated Data

IncludeAttribute can also be used for another purpose – **data denormalization**. For example, assume that in the above scenario we don’t really need the entire ProductSubCategory for each Product, we only want the Name of the category.

internal sealed class ProductMetadata

{

[Include("Name", "SubCategoryName")]

public ProductSubcategory ProductSubcategory;

}

The above attribute results in a “SubCategoryName” member being generated on the client Product entity. When querying Products, only the category Name is sent to the client, not the entire category. Note that the ProductSubCategory must still be loaded from the DAL as it was in the previous example.

## Query Operators

EntityQuery exposes only a subset of LINQ query operators. These are:

1. Where
2. OrderBy / ThenBy
3. Skip / Take

## POST support for Query operations

By default .NET RIA Services over HTTP uses a GET request for query operations. POST is also supported and can be indicated per query operation by specifying **HasSideEffects=true** in the QueryAttribute applied to the operation.

## Supported Types

This section outlines the list of types currently supported by the .NET RIA Services, and such type restrictions pertain to:

* The types of Entity members exposed by all domain operations
* The types of return parameters of ServiceOperations
* The types of optional method parameters of Query, Custom or Service Operations

### Simple Supported Types

The following simple types are supported:

* .NET Framework primitive types[[11]](#footnote-11):
  + Boolean
  + Byte
  + SByte
  + Int16
  + UInt16
  + Int32
  + UInt32
  + Int64
  + UInt64
  + IntPtr
  + UIntPtr
  + Char
  + Double
  + Single
* String
* Decimal
* DateTime
* TimeSpan
* Guid
* Uri
* XElement

### Complex Supported Types

The following complex types are supported:

* Byte[]
* System.Data.Linq.Binary[[12]](#footnote-12)
* Array of one of the simple supported types listed above
* IEnumerable<T> where T is one of the simple supported types listed above
* Nullable<T> where T is one of the simple supported types listed above
* Any types that implements IList AND defines an empty default constructor

# How to Modify Entities

This section explores the client/server concepts and APIs related to performing insert/update/delete operations on application data. If you’ve followed the introductory .NET RIA Services walkthrough (installed with the preview), you’ve seen how to update data using a DataForm. This section will explore the underlying APIs that such data bound controls are built on. In many cases, you’ll simply let the controls do most of the work for you, but for advanced scenarios or simply for curiosity sake you may need to understand what is going on under the covers. In this walkthrough we’ll create a simple driver application for update operations, and explore the relevant features and APIs.

## Sample Setup

To start, follow the walkthrough steps to create a sandbox application that we can use to explore the APIs:

* Create the new Silverlight application, naming it “UpdateSample”, configuring all the other project settings as in the walkthrough
  + Make sure the SL app has a reference to the System.Windows.Controls.Data assembly
* Add a copy of the AdventureWorks DB referenced in the walkthrough to the App\_Data folder of the Web project
* Add a new LINQ to SQL data model to the Web project by selecting “LINQ to SQL classes” in the “Add New” dialog
  + Name the DBML file “AdventureWorks.dbml”
  + Use Server Explorer to add a new Data Connection to the AdventureWorks\_Data.mdf file
* Use LINQ to SQL designer to add the Product entity, by dragging the Product table from Server Explorer onto the designer surface
* Build the solution
* Add a new DomainService class to the web app by following the steps under “Creating a DomainService” in the .NET RIA Services overview document
  + Name the DomainService “Catalog”
  + Select the Product entity and make sure the “Enable Editing” checkbox is checked
  + Select the “Generate associated classes for metadata” checkbox
* Build the solution
* Finally bind the UI and load some data by adding the following XAML and code-behind (new code highlighted):

<UserControl x:Class="UpdateSample.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"

Width="Auto" Height="Auto">

<Grid x:Name="LayoutRoot">

<data:DataGrid Height="Auto" Width="Auto" x:Name="productsGrid"></data:DataGrid>

</Grid>

</UserControl>

using UpdateSample.Web;

using System.Windows.Ria.Data;

namespace UpdateSample {

public partial class MainPage : UserControl {

Catalog \_catalog = new Catalog();

public MainPage() {

InitializeComponent();

productsGrid.ItemsSource = \_catalog.Products;

var query = from p in \_catalog.GetProductsQuery()

where p.ProductSubcategoryID == 1

select p;

\_catalog.Load(query);

}

}

}

## Updating entities

When this simple data bound app runs it queries and displays the Product data in the grid. Since the grid’s ItemsSource is bound to the Products EntityList, when the asynchronous query returns and the returned entities are loaded into the Catalog.Entities EntityList, the grid will automatically update. Below we’ll explore some update concepts, and add update functionality to the app.

### DomainService Update Methods

A DomainService is where your server business logic is written. The service exposes one or more entities and a set of operations for that entity, and during build a corresponding client representation is automatically generated into your Silverlight application. For example, assume you’ve used the DomainService class wizard to generate the below DomainService class exposing the Product entity. The generated service class has Create/Read/Update/Delete (CRUD) operations for Product:

[EnableClientAccess()]

public class Catalog :

LinqToSqlDomainService<AdventureWorksDataContext>

{

public IQueryable<Product> GetProducts()

{

return this.Context.Products;

}

public void InsertProduct(Product product)

{

this.Context.Products.InsertOnSubmit(product);

}

public void UpdateProduct(Product currentProduct)

{

this.Context.Products.Attach(currentProduct,

this.ChangeSet.GetOriginal(currentProduct));

}

public void DeleteProduct(Product product)

{

this.Context.Products.Attach(product);

this.Context.Products.DeleteOnSubmit(product);

}

}

These methods, in addition to declarative validation metadata, are the focal point for your business logic development. Note that the above code uses naming convention to indicate the operations – if you don’t wish to follow naming conventions, you can use declarative attributes (InsertAttribute/UpdateAttribute/DeleteAttribute/QueryAttribute/ResolveAttribute) to mark your methods. See the separate section on convention/configuration for more details, but the naming conventions are:

* **Insert**
  + “Insert”, “Add”, “Create” prefixes
  + Method signature : void InsertX(T entity), where T is an entity Type
* **Update**
  + “Update”, “Change”, “Modify” prefixes
  + Method signature : void UpdateX(T current), where T is an entity Type
* **Delete**
  + “Delete”, “Remove” prefixes
  + Method signature : void DeleteX(T entity), where T is an entity Type
* **Query**
  + Method signature returning a singleton, IEnumerable<T> , IQueryable<T> where T is an entity Type, and taking zero or more parameters
* **Resolve**
  + “Resolve” prefix
  + Method signature : bool ResolveX(T curr, T original, T store, bool isDelete) where T is an entity type

The DomainService operations present for each entity dictates the client code that is generated and what operations that are permitted. In this example, you’ll see that the generated Catalog DomainContext contains the following code:

protected override EntityContainer CreateEntityContainer()

{

return new CatalogEntityContainer();

}

internal sealed class CatalogEntityContainer : EntityContainer

{

public CatalogEntityContainer()

{

this.CreateEntityList<Product>(EntityListOperations.All);

}

}

This code creates the EntityContainer that will be used by the client, along with an EntityList for each entity type exposed, configured with the correct operation types supported by the service. In this case, EntityListOperations. “EntityListOperations.All” indicates that query and all update operations are supported for Product. If only a subset of the CRUD operations were supported by the service, this generated code would reflect that.

## EntityContainer, EntityList and Change Tracking

Entities loaded into a DomainContext are stored and change tracked by the EntityContainer associated with the DomainContext (DomainContext.Entities). An Entity container manages a set of EntityLists, and there is a 1:1 mapping between EntityLists and the entity Types exposed by your service. EntityContainer and its EntityLists comprise a collaborative whole which is responsible for all identity management and change tracking on the client. A DomainContext always has a single EntityContainer it uses, but it also provides some APIs which simply delegate to the underlying container (e.g. HasChanges, RejectChanges, etc.). Inspection of the Entity.EntityState of the loaded entities will yield EntityState.Unmodified for all unmodified entities. Any edits to such entities, as well as adding new entities and deleting existing entities will also be change tracked, and the entity’s EntityState will reflect that.

To explore this, modify the XAML and code-behind as below to add some simple update functionality to the app:

<UserControl x:Class="UpdateSample.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"

Width="Auto" Height="Auto">

<Grid x:Name="LayoutRoot">

<StackPanel Orientation="Vertical" Height="Auto" Grid.Column="0">

<StackPanel Orientation="Horizontal">

<Button x:Name="saveButton" Width="75" Height="30" Content="Save" Margin="5" Click="saveButton\_Click"/>

<Button x:Name="rejectButton" Width="75" Height="30" Content="Reject" Margin="5" Click="rejectButton\_Click"/>

<TextBlock x:Name="changeText" VerticalAlignment="Center" Width="Auto"/>

</StackPanel>

<data:DataGrid Height="Auto" Width="Auto" x:Name="productsGrid"></data:DataGrid>

</StackPanel>

</Grid>

</UserControl>

public partial class MainPage : UserControl

{

Catalog \_catalog = new Catalog();

public MainPage()

{

InitializeComponent();

UpdateHasChanges();

\_catalog.PropertyChanged += (s, e) =>

{

if (e.PropertyName == "HasChanges")

{

UpdateHasChanges();

}

};

productsGrid.RowEditEnded += productsGrid\_RowEditEnded;

productsGrid.KeyUp += productsGrid\_KeyUp;

productsGrid.ItemsSource = \_catalog.Products;

var query = from p in \_catalog.GetProductsQuery()

where p.ProductSubcategoryID == 1

select p;

\_catalog.Load(query);

}

void productsGrid\_RowEditEnded(object sender,

DataGridRowEditEndedEventArgs e)

{

// every time a row is updated, update the change state

UpdateHasChanges();

}

void productsGrid\_KeyUp(object sender, KeyEventArgs e)

{

if (e.Key == Key.Delete && productsGrid.SelectedItem != null)

{

Product product = (Product)productsGrid.SelectedItem;

\_catalog.Products.Remove(product);

UpdateHasChanges();

}

}

private void UpdateHasChanges()

{

// get the current changeset from the EntityContainer

EntityChangeSet changeSet = \_catalog.Entities.GetChanges();

changeText.Text = changeSet.ToString();

// update the button states

bool hasChanges = \_catalog.HasChanges;

saveButton.IsEnabled = hasChanges;

rejectButton.IsEnabled = hasChanges;

}

private void rejectButton\_Click(object sender, RoutedEventArgs e)

{

\_catalog.RejectChanges();

}

private void OnSubmitCompleted(SubmitOperation so)

{

if (so.Error != null)

{

MessageBox.Show(

string.Format("Submit Failed : {0}", so.Error.Message));

}

}

private void saveButton\_Click(object sender, RoutedEventArgs e)

{

\_catalog.SubmitChanges(OnSubmitCompleted, null);

}

}

Build and run the application, and make a few updates to entities in the grid, for example changing a Color and some ListPrices. You can see in the UI that the updates are being tracked (the Modified count is incremented on each edit). These updates are tracked locally and aren’t submitted to the server until DomainContext.SubmitChanges is called.

## Change Sets (Unit of Work)

A change set is a set of one or more entity operations to be processed as a single unit of work. A change set can contain:

* Entity Insert/Update/Delete operations
* Custom domain operations (e.g. ApprovePurchaseOrder)

As the client makes changes to entities, they are tracked by the EntityContainer, and when SubmitChanges is called the changes are all sent to the service all together. When processed by the service, the change set succeeds or fails as a single unit.

### Client Side Change Set Processing

In the UpdateHasChanges event handler, you can see the line:

EntityChangeSet changeSet = catalog.Entities.GetChanges();

At any time, the current change set can be computed by calling EntityContainer.GetChanges. The collections on the change set contain all the entity operations currently pending. To discard all pending changes, simply call DomainContext.RejectChanges(). You can see this in the code:

private void rejectButton\_Click(object sender, RoutedEventArgs e)

{

catalog.RejectChanges();

}

Hitting the Reject button with pending changes will cause RejectChanges to be called, all changes will be discarded, and the bound UI will update itself.

Hitting the Save button will cause SumitChanges to be called. Set a breakpoint on the UpdateProduct method in the server Catalog DomainService and you’ll see that when changes are submitted, your update method is called for each updated Product. The client change set is serialized and sent to the server and the update operations are played back through their corresponding update methods. For details on the **SubmitOperation** Type returned from SubmitChanges and the functionality it provides, as well as how to specify a submit callback, see the section “Asynchronous Domain Operations”.

To test deletes, add the following Delete key handler code to the code-behind, and attach the handler to the grid’s KeyUp event:

void productsGrid\_KeyUp(object sender, KeyEventArgs e)

{

if (e.Key == Key.Delete && productsGrid.SelectedItem != null)

{

Product product = (Product)productsGrid.SelectedItem;

\_catalog.Products.Remove(product);

UpdateHasChanges();

}

}

You can now delete rows in the grid and the corresponding service delete operations will be called. However in this sample the deletes will likely fail due to FK constraints in the DB. Similar to updates and deletes, you can add UI for adding new rows, making the corresponding EntityList.Add (catalog.Products.Add) call for the new entities, and you’ll see that the new entities are part of the change set as well.

### Server Side Change Set Processing

When a change set is sent to the DomainService, the sequence of operations on the DomainService is:

1. **Submit** – the service entry point that receives the change set and begins the change set processing pipeline
2. **AuthorizeChangeSet** – Verifies Permission/Authorization for each operation. If any operations are not permitted, processing stops and an error is returned to the client
3. **ValidateChangeSet** – Validation for all operations in the change set is run. If any operations fail validation, processing stops and errors are returned to the client
4. **ExecuteChangeSet** – The corresponding method for each operation in the change set is invoked. This is the point in time when your business logic is run. If any operations failed, processing stops and errors are returned to the client
5. **PersistChangeSet** – At this point all the domain operations have been executed, and this method is called to finalize the changes.
6. **ResolveChangeSet** – If there were any optimistic concurrency errors, this method is called to resolve them.

You’ll see that each of these operations is virtual and can be overridden to participate in the update pipeline. Similarly, on the client DomainContext’s Load/Invoke/Submit operations are also virtual allowing for pre/post handling.

## Error Handling

When submitting changes it is important to check the Error property of the SubmitOperation returned SubmitChanges operation completes to determine if the operation succeeded and to handle any errors. Modify the OnSubmitCompleted handler as below:

private void OnSubmitCompleted(SubmitOperation so)

{

if (so.Error != null)

{

string message = so.Error.Message;

if (so.EntitiesInError.Any())

{

message = string.Empty;

Entity entityInError = so.EntitiesInError.First();

if (entityInError.Conflict != null)

{

EntityConflict conflict = entityInError.Conflict;

foreach (EntityConflictMember cm in

conflict.MemberConflicts)

{

message += string.Format(

"Member '{0}' in conflict: Current: {1},

Original: {2}, Store: {3}",

cm.PropertyName, cm.CurrentValue,

cm.OriginalValue, cm.StoreValue);

}

}

else if (entityInError.ValidationErrors.Any())

{

message += "\r\n" +

entityInError.ValidationErrors.First().Message;

}

}

MessageBox.Show(message, "Submit Failed", MessageBoxButton.OK);

}

}

Add a validation check in the DomainService UpdateProduct method like this:

public void UpdateProduct(Product currentProduct)

{

if (currentProduct.ListPrice == 0)

{

throw new ValidationException(

"List price must be greater than zero!");

}

this.Context.Products.Attach(currentProduct,

this.ChangeSet.GetOriginal(currentProduct));

}

Exercise this code by building and running the app, and attempting to update a ListPrice to zero and Saving the changes. The error information is returned to the client and can be displayed to the user for correction. This simple example isn’t realistic, but shows how to check for errors and access error information on the client. In real world apps, data bound controls will be used to interpret and present such information in a rich way. For example, DataForm checks for and interprets such errors and provides member level error UI on the input controls.

## Concurrency Conflicts

Concurrency conflict errors are reported to the client in the same way described above. As you can see in the above code, concurrency conflicts returned to the client are reported on the individual entities in the failed change set via their Entity.Conflict member. To test this code, simply open up two separate instances of the app in the browser. After both have loaded, generate a concurrency conflict by using one app to update the ListPrice of the first Product to a new value and submit the changes. In the second app, attempt to update another member like Color. Attempting to submit that update will result in a concurrency exception with the detailed conflict error info being returned to the client.

You can add type specific Resolve methods to your DomainService to specify server side conflict resolution logic that will be called when conflicts for that type occur. For example, add the following method to your Catalog DomainService to resolve Product conflicts:

public bool ResolveProduct(

Product currentProduct, Product originalProduct,

Product storeProduct, bool deleteOperation)

{

return base.Resolve(currentProduct, originalProduct, storeProduct,

ResolveOption.KeepChanges);

}

This method calls into the base LinqToSqlDomainService Resolve helper method which will resolve the conflict according to the ResolveOption specified. Set a breakpoint and run the same conflict scenario above. You’ll see that your Resolve method is called to resolve the conflict server side and a conflict is no longer received by the client. Refreshing the view shows that the updated value was persisted to the DB.

## Transactions

The framework itself doesn’t create any transactions, but it does provide hooks in the update processing pipeline for you to manage your own. In this example, we’re using a LINQ to SQL (LTS) DAL as the backing store, so the DomainService.PersistChangeSet implementation for the LinqToSqlDomainService calls SubmitChanges on the underlying LTS DataContext. At that point all the update operations have been performed, and the transactional behavior of LTS DataContext is leveraged; if there isn’t an ambient transaction in scope, an implicit transaction is created, and otherwise LTS will enlist in the existing transaction.

To create your own explicit transaction, you can override DomainService.Submit and wrap the call to the base method in your own transaction scope. For example:

public override void Submit(

System.Web.DomainServices.ChangeSet changeSet)

{

using (var tx = new TransactionScope(

TransactionScopeOption.Required,

new TransactionOptions

{ IsolationLevel = IsolationLevel.ReadCommitted })

)

{

// call to base to process the changeset within

// our transaction scope

base.Submit(changeSet);

// complete the transaction

tx.Complete();

}

}

With the above override, all operations performed by the domain operations called during change set processing will be done under this explicitly created transaction scope.

## Identity Management

EntityContainer is responsible for identity tracking for entities loaded on the client. Every time an entity is loaded or attached to the container, the instance is cached by the identity value returned from Entity.GetIdentity. You can see this in the generated code for Product (view by selecting “Go To Definition” on the Product Type):

public override object GetIdentity()

{

return this.\_productID;

}

By default, whenever an entity is loaded if there is a cached instance locally with the same identity or key value(s) the local instance will be retained, and the newly read instance is discarded. This ensures there is ever only a single instance in the container for each unique identity. This also ensures for example that entity modifications are preserved when data is reloaded. This behavior can be modified by using the Load method overload that takes a MergeOption. For example, if you want to refresh the client data, you can issue a load call specifying MergeOption.KeepChanges:

\_catalog.Load(query, MergeOption.KeepChanges);

Doing this merges in any product updates made in the database, while retaining any local updates. Similarly, to discard any local updates and update to the database values, MergeOption.OverwriteCurrentValues can be used. The default is MergeOption.KeepCurrentValues.

## IEditableObject/IChangeTracking/IEditableCollection

There are several interfaces related to updates and change tracking that Types in the framework implement. These are general interfaces that data controls use to provide update functionality for objects.

### IEditableObject

Entity explicitly implements IEditableObject, which allows a set of entity property modifications to be made transactionally.

namespace System.ComponentModel {

public interface IEditableObject {

void BeginEdit();

void CancelEdit();

void EndEdit();

}

The entity takes a “snapshot” of itself whenever BeginEdit is called, and if CancelEdit is subsequently called, all changes made since the edit session was started are reverted. Similarly, EndEdit is called to complete the edit session and accept the changes. Data controls like DataForm interact with Entity through this interface. For example, hitting the edit button of a DataForm bound to an Entity causes DataForm.BeginEdit to be called, and which in turns calls the corresponding method on Entity. The same goes for cancelling or committing the changes made in the DataForm. Take a look at the following code:

// in this session, the changes are persisted

Product product = catalog.Products[0];

((IEditableObject)product).BeginEdit();

product.ListPrice += 3.14M;

product.Color = "Purple";

((IEditableObject)product).EndEdit();

// in this session, the changes are rolled back

((IEditableObject)product).BeginEdit();

product.Color = "Blue";

((IEditableObject)product).BeginEdit(); // has no effect

product.ReorderPoint = 5;

((IEditableObject)product).CancelEdit();

### IChangeTracking/IRevertibleChangeTracking

Entity, EntityList and EntityContainer all implement these interfaces explicitly.

namespace System.ComponentModel {

public interface IChangeTracking {

bool IsChanged { get; }

void AcceptChanges();

}

public interface IRevertibleChangeTracking : IChangeTracking {

void RejectChanges();

}

}

These interfaces are used during client change set processing and are not intended to be used directly by application developers. As with IEditableObject, these are the interfaces general data controls like DataForm use to interact with the objects they’re bound to (without depending on the EntityType). For example, DataForm uses IChangeTracking.IsChanged to determine if a bound entity has been modified. Entity, EntityList and EntityContainer have a HasChanges member that is public, which delegates to the IsChanged implementation.

### IEditableCollection/IEditableCollection<T>

EntityList explicitly implements IEditableCollection. Controls like DomainDataSource and DataForm interact with an EntityList through this interface to determine which operations are supported (based on the operations supported by the DomainService), as well as to perform edit operations.

namespace System.ComponentModel {

public interface IEditableCollection {

bool CanAdd { get; }

bool CanEdit { get; }

bool CanRemove { get; }

void Add(object item);

void BeginEdit(object item);

void CancelEdit(object item);

object CreateNew();

void EndEdit(object item);

void Remove(object item);

}

}

# Using Custom Methods and Service Operations

Previously, we have shown how to implement simple data manipulation. In this section we explain how to implement useful application logic in addition to the basic create/update/delete operations. In order to illustrate new concepts, we will build an inventory application that shows our catalog together with competitor’s prices and allows us to discount products.

## Sample Setup

To start, follow these simple steps to create a sandbox application that we can use to explore the APIs:

* Create a new Silverlight application, naming it “CustomSample” and enable .NET RIA Services
* Make sure the SL app has a reference to the System.Windows.Controls.Data assembly
* Add a copy of the AdventureWorks database to the App\_Data folder of the Web project
* Add a new LINQ to SQL data model to the Web project by selecting “LINQ to SQL classes” in the Add 🡪New Item dialog
* Use LINQ to SQL designer to add the Product entity, by dragging the Product table from Server Explorer onto the designer surface
* Build the solution
* Add new DomainService for your data model by using Add🡪New Item🡪Web🡪Domain Service
  + Name the DomainService “Catalog”
  + Select the Product entity and make sure the “Enable Editing” checkbox is checked
  + Select the “Generate associated classes for metadata” checkbox
* Build the solution
* Create the UI that loads data by adding the following XAML and code-behind:

<!-- MainPage.xaml -->

<UserControl xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"

x:Class="CustomSample.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

Width="Auto" Height="Auto">

<Grid x:Name="LayoutRoot" Background="White">

<StackPanel Orientation="Vertical">

<data:DataGrid IsReadOnly="True" Name="productsGrid"

Height="Auto" Width="Auto"

SelectionChanged="DataGrid\_SelectionChanged"/>

<StackPanel Orientation="Horizontal">

<TextBlock Text="Competitor's price" Margin="2"/>

<TextBlock Name="compPrice" Margin="2"/>

</StackPanel>

<StackPanel Orientation="Horizontal">

<Button Click="Button\_Click" Content="Discount"

Name="discountButton" Margin="2"/>

<TextBox Name="discountPercent" Text="10" Margin="2,2"/>

<TextBlock Text="%" Margin="2,5,0,2"/>

</StackPanel>

</StackPanel>

</Grid>

</UserControl>

// MainPage.xaml.cs

using System.Windows;

using System.Windows.Controls;

using System.Windows.Ria.Data;

using CustomSample.Web;

namespace CustomSample

{

public partial class MainPage : UserControl

{

Catalog catalog = new Catalog();

public MainPage()

{

InitializeComponent();

productsGrid.ItemsSource = catalog.Products;

var query = from p in catalog.GetProductsQuery()

where p.ListPrice > 3000

select p;

catalog.Load(query);

}

private void Button\_Click(object sender, RoutedEventArgs e)

{

Product selectedProduct = (Product)productsGrid.SelectedItem;

if (selectedProduct != null)

{

int percentage = int.Parse(discountPercent.Text);

//selectedProduct.DiscountProduct(percentage);

//catalog.SubmitChanges();

}

}

private void DataGrid\_SelectionChanged(object sender, SelectionChangedEventArgs e)

{

Product selectedProduct = (Product)productsGrid.SelectedItem;

if (selectedProduct != null)

{

//catalog.GetCompetitorsPrice(selectedProduct,

// (invokeOperation) =>

// {

// compPrice.Text = invokeOperation.Value.ToString();

// }, null);

}

}

}

}

## Writing Application Logic

Now that UI is done, we will add the ability to discount the selected product and display competitor’s price. In order to discount the selected product we need to lower its price by a percentage that the user entered. In order to display competitor’s price we need to do an out of band server lookup. In a real app, this lookup might be fairly complicated such as querying another site to get the best price. For simplicity we’ll just return a price that’s 5% lower than the list price.

### Custom Methods

A Custom method is an asynchronous domain operation that is used when some extra application logic needs to be performed in addition to basic create/update/delete operations. For example, custom method could do some logging or reporting after the operation has been carried out.

Custom methods have change tracking and deferred execution like the basic create/update/delete operations. This means that changes will not be submitted to the server until the DomainContext.SubmitChanges() method is called (see “How to modify entities” section for more details).

#### Declaration

There are two ways to mark a method as a custom method inside a DomainService:

* By convention a method that returns void, takes an entity as the first parameter and is not a create/update/delete method is considered a custom method.
* By applying CustomAttribute to the method.

We illustrate both ways (only one is necessary) in the following code snippet that implements the discount operation that we talked about earlier. This code should be added to the “Catalog” DomainService.

[Custom]

public void DiscountProduct(Product product, int percentage)

{

this.Context.Products.Attach(product);

decimal newPrice = product.ListPrice \* (1 - percentage / 100m);

product.ListPrice = newPrice;

// Do logging/reporting here

}

RIA Services passes entity parameters to custom methods in their current state. If you need to access the original state, use:   
this.ChangeSet.GetOriginal(product);

#### Usage

Custom methods can be called from the client in two ways:   
using the generated Domain Context and passing in the entity  
catalog.DiscountProduct(selectedProduct, percentage);  
or on the entity itself  
selectedProduct.DiscountProduct(percentage);

### Service Operations

Compared to the other DomainService operation types discussed above, service operations more closely resemble traditional web methods. They can be used in situations when the server-side logic does not fit create/update/delete pattern. Unlike the other DomainService operations they do not have change tracking or deferred execution. This means that the Service operation call will be submitted to the server as soon as the corresponding method call is made on the client.

#### Declaration

There is no naming convention for marking a method as a service operation. To mark a method as a service operation, use the ServiceOperation attribute. Here’s a sample service operation that gets competitor’s price for given product. This code should be added to the “Catalog” DomainService:

[ServiceOperation]

public decimal GetCompetitorsPrice(Product product)

{

// Do some kind of price lookup.

return product.ListPrice \* 0.95m;

}

The parameters and return type of ServiceOperation must be an entity or one of the predefined serializable types.

#### Usage

The important thing to note here is that like the rest of our framework, service operation execution is asynchronous. There are three ways to access service operation’s return value. Developer can:   
bind directly to InvokeOperation.Value like this:

InvokeOperation<decimal> invokeOp = catalog.GetCompetitorsPrice(selectedProduct);

compPrice.Text = invokeOp.Value.ToString();

pass a callback function like this:

catalog.GetCompetitorsPrice(selectedProduct,

(invokeOperation) =>

{

compPrice.Text = invokeOperation.Value.ToString();

}, null);

or register an event handler like this:

InvokeOperation<decimal> invokeOp = catalog.GetCompetitorsPrice(selectedProduct);

invokeOp.Completed += new System.EventHandler(invokeOp\_Completed);

void invokeOp\_Completed(object sender, System.EventArgs e)

{

InvokeOperation<decimal> invokeOp = (InvokeOperation<decimal>)sender;

compPrice.Text = invokeOp.Value.ToString();

}

## Conclusion

Now you can uncomment the code in MainPage.xaml.cs that calls DiscountProduct() and GetCompetitorsPrice() and run the application. Select a product in the DataGrid and competitor’s price will appear; you can then mark the product down by using “Discount” button.

# Understanding Convention and Configuration

## Motivation behind convention-based design

Convention-based approaches:

1. Require less work for the end developer
2. Provide a more consistent and esthetically pleasing experience
3. Provide a common domain-driven language and encourage readability of code across teams and organizations

## Rules that apply to all DomainOperations

* DomainOperations must have public modifier
* All but one DomainOperations can only exist if their first (or first several) parameter types are entities. When a query method returns an IEnumerable/IQueryable of a type, that type becomes an entity as far as RiaServices is concerned. Therefore a Query DomainOperation must be present for other DomainOperations to be valid. The exception to the rule is ServiceOperations which do not require a an entity type to be valid DomainOperations.
* DomainOperations must use serializable types for their parameters and return types. See the section “Supported Types” for more information.

## DomainOperations

### Insert

Convention requires: Name and signature

Signature: void returning method with an entity as the only parameter

Name: Prefix must be one of the following: “Insert” | “Add” | “Create”

Example:

public void InsertEmployee(Employee newEmployee) {...}

Any name can be used if the **InsertAttribute** is applied:

[Insert]

public void YourFavoriteMethodName(Employee newEmployee) {...}

### Update

Convention requires: Name and signature

Signature:  void returning method with an entity as the only parameter

Name: Prefix must be one of the following: “Update” | “Change” | “Modify”

Example:

public void UpdateEmployee(Employee changedEmployee)

Any name can be used if the **UpdateAttribute** is applied:

[Update]

public void YourFavoriteMethodName(Employee changedEmployee)

### Delete

Convention requires: Name and signature

Signature: void returning method with an entity as the only parameter

Name: Prefix must be one of the following: “Delete” | “Remove”

Example:

public void DeleteEmployee(Employee currentEmployee)

Any name can be used if the **DeleteAttribute** is applied:

[Delete]

public void YourFavoriteMethodName(Employee currentEmployee)

### Query

Signature match only: Any method returning IEnumerable<T>, IQueryable<T> or T where T is an entity.

public IQueryable<Employee> GetEmployee() {...}

//Can choose to return a single entity rather than IQueryable/IEnumerable

public City GetCity() {...}

You can explicitly indicate a Query DomainOperation using the **QueryAttribute**:

[Query]

public IQueryable<Employee> GetEmployee() {...}

### Custom

Signature match only: void returning method with entity as the first parameter that is not an Insert, Update or Delete DomainOperation

 public void ApproveEmployee(Employee changedEmployee, ...)

You can explicitly create a Custom DomainOperation using the **CustomAttribute**:

[Custom]

public void ApproveEmployee(Employee changedEmployee, ...)

### Resolve

Convention requires: Name, signature and Update DomainOperation exists

Signature: bool returning method, first three parameters must be the entity type, forth parameter a Boolean. The order of the three entity parameters is: current, original, and store.

Name: Prefix must be “Resolve”

Example:

public bool ResolveEmployee(PurchaseOrder currentEmployee, PurchaseOrder originalEmployee, PurchaseOrder storeEmployee, bool deleteOperation)

It is important to remember that Resolve DomainOperations require an Update DomainOperation.

### ServiceOperation

No convention; must use ServiceOperationAttribute:

[ServiceOperation]

public byte[] GetProductImage(){...}

### IgnoreOperationAttribute

When using the convention based approach, sometimes it might be necessary to explicitly exclude a member from the inference through convention process. For example, you might have a public member that conforms to one of the operation conventions, but you don’t want it to be inferred as an operation. In such cases, you can apply **IgnoreOperationAttribute** to the member and it will be ignored.

For example, the following InsertEmployee method will be ignored (and client inserts will not be available) even though it matches the convention requirements for an Insert DomainOperation:

[IgnoreOperation]

public void InsertEmployee(Employee newEmployee) {...}

IgnoreOperationAttribute can be applied to all DomainOperations that use the convention-based approach.

# Asynchronous Domain Operations

DomainContext supports three kinds of domain operations : **Query, Submit and Invoke**. Each of these operations has a corresponding operation Type that represents the in-progress asynchronous operation. For example, when a query is loaded into a DomainContext, the asynchronous Load call returns a **LoadOperation** instance immediately:

HRContext \_ctxt = new HRContext();

var query = from e in \_ctxt.GetEmployeesQuery()

where e.ManagerID == 1

select e;

LoadOperation<Employee> lo = \_ctxt.Load(query);

The operation result provides both bindable dynamic status information on the progress of the operation as well as cancelation capability. Similarly, a submit request returns a SubmitOperation, and a service operation invocation returns an InvokeOperation. Below, these operations and their base class are discussed in more detail.

## OperationBase

This is the base class for all .NET RIA Services asynchronous operations. It implements INotifyPropertyChanged, so it’s operation state members can be data bound and notifications will be raised when the operation completion state changes.

* **IsComplete** – returns true once the operation has completed. An operation is considered complete when it has completed successfully, has completed with an error, or has been canceled.
* **Completed event** – This event is raised when the operation completes.
* **UserState** – optional user state associated with the operation.
* **Cancel () / CanCancel / IsCanceled** – CanCancel returns true if the operation is canceleable and hasn’t completed yet. CanCancel should be checked before calling Cancel. If the operation has already completed, an exception will be thrown.
* **Error / HasError** **–** if an operation has completed with error, HasError will be true and Error will contain the Exception.

As you’ll see below, an operation can also have a **completion callback** specified. If a callback has been specified, it will be called in all cases, regardless of whether the operation has completed successfully, has completed with error or has been canceled. Completion callbacks are also invoked before the Completed event is raised.

## LoadOperation

DomainContext.Load<T> returns a LoadOperation<T> instance. LoadOperation represents an asynchronous load operation in progress, and exposes the following additional state and functionality:

* **Entities** – A strongly typed enumeration of the entities loaded. Immediately after the Load call returns, the collection will be empty. The collection implements INotifyCollectionChanged, and will update itself when the operation completes.
* **AllEntities** – A collection of all entities loaded by the query operation, including any associated entities. Also implements INotifyCollectionChanged. Immediately after the Load call returns, the collection will be empty. The collection implements INotifyCollectionChanged, and will update itself when the operation completes.
* **TotalEntityCount** – returns the total server entity count for the query.
* **EntityQuery** – the EntityQuery used to initiate the load operation.
* **MergeOption** – the MergeOption specified for the load operation.

The below code shows how to specify a load callback:

object state = . . . // user state

LoadOperation<Employee> lo = \_ctxt.Load(query,

EmployeesLoadedCallback, state);

. . .

private void EmployeesLoadedCallback(LoadOperation<Employee> lo)

{

if (!lo.HasError)

{

object state = lo.UserState;

foreach (Employee employee in lo.Entities)

{

// handle the loaded entities

}

}

else

{

// handle error

}

}

To specify a completion event handler, LoadOperation.Completed can be subscribed to as below. This is useful in scenarios where multiple (possibly external) parties may be interested in the operation progress.

object state = . . . // user state

LoadOperation<Employee> lo = \_ctxt.Load(query, null, state);

lo.Completed += this.OnEmployeesLoaded;

. . .

private void OnEmployeesLoaded(object sender, EventArgs e)

{

LoadOperation<Employee> lo = (LoadOperation<Employee>)sender;

. . .

}

## SubmitOperation

DomainContext.SubmitChanges returns a SubmitOperation instance. SubmitOperation represents the asynchronous submit operation in progress, and exposes the following additional state and functionality:

* **ChangeSet** – The EntityChangeSet that is being submitted.
* **EntitiesInError** – For a submit operation that has completed with validation or conflict errors, this is a collection of the entities in error.

Callbacks and Completed event subscription works the same as for LoadOperation which was explained above.

## InvokeOperation

When a ServiceOperation is invoked, a strongly typed InvokeOperation<T> is returned, where T is the return type of the ServiceOperation. InvokeOperation exposes the following additional state and functionality:

* **OperationName** – The name of the operation invoked
* **Parameters** – Dictionary of the name/values of the parameters to the operation
* **ValidationErrors** – Collection of the validation errors for a failed invocation
* **Value** – the value returned by the ServiceOperation.

Callbacks and Completed event subscription works the same as for LoadOperation which was explained above.

# Using the Silverlight DomainDataSource

This section explores how to use the Silverlight DomainDataSource to work with data supplied by a DomainContext. The DomainDataSource completes the Silverlight client side of the .NET RIA Services story that begins with Entities and DomainServices, undergoes code-generation to produce Entities and DomainContexts, and finally surfaces that data to a rich UI. With the DomainDataSource, it is possible to declare the interactions between your UI and the data with minimal code-behind.

## Sample Setup

To start, follow these steps to create a sandbox application for working with the DomainDataSource:

* Select File🡪New Project🡪Silverlight Application
* In the resulting dialog, ensure that “Host the Silverlight application in a new Web site” and “Enable .NET RIA Services” are checked, with “ASP.NET Web Application Project” selected as the New Web project type
* Add references in your Silverlight Application to the following assemblies:
  + System.Windows.Controls.Data
  + System.Windows.Ria.Controls
* To the ASP.NET server project that has been created, add “LINQ to SQL Classes” (Project🡪Add New Item…). Call it “AdventureWorks.dbml”
* Drag the Employee table from the AdventureWorks database (used in the walkthrough) onto the design surface.
* Build your solution.
* To the ASP.NET server project, add a “Domain Service Class” called AdventureWorksDomainService.cs.
  + In the dialog that appears, check both of the boxes next to Employee and click OK to generate full CRUD support for the provider.
  + Add the following code to the AdventureWorksDomainService class:

public IQueryable<Employee> GetEmployeesByGender(char gender)

{

return this.Context.Employees.Where((emp) => emp.Gender.Equals(gender));

}

* Build your solution.
* Add the following xmlns definitions to the User Control in MainPage.xaml:

xmlns:riaControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Ria.Controls"

xmlns:riaData="clr-namespace:System.Windows.Data;assembly=System.Windows.Ria.Controls"

xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"

* Copy the following code into the UserControl in MainPage.xaml (do not replace the root-level tag):

<Grid>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<data:DataGrid />

<Grid Grid.Row="1">

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<data:DataPager Grid.Row="0" Grid.ColumnSpan="2" />

<TextBlock Grid.Row="1" Text="LoginID" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBlock Grid.Row="2" Text="ContactID" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBlock Grid.Row="3" Text="Gender" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBlock Grid.Row="4" Text="Minimum Vacation Hours" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBox x:Name="loginID" Grid.Row="1" Grid.Column="1" Margin="4" />

<TextBox x:Name="contactID" Grid.Row="2" Grid.Column="1" Margin="4" />

<ComboBox x:Name="gender" Grid.Row="3" Grid.Column="1" Margin="4" SelectedIndex="1">

<ComboBoxItem Content="M" />

<ComboBoxItem Content="F" />

</ComboBox>

<TextBox x:Name="vacationHours" Grid.Row="4" Grid.Column="1" Margin="4" />

</Grid>

</Grid>

* Finally, you will need to add an xmlns definition for your generated DomainContext. Your xmlns will look like this, replacing <YourSilverlightApplicationNamespace> with the name of your Silverlight project:

xmlns:domain="clr-namespace:<YourSilverlightApplicationNamespace>.Web"

The XAML above will provide some simple UI that we will use with the DomainDataSource. Specifically, it includes a DataGrid for displaying the requested data and a number of TextBoxes and ComboBoxes for adjusting the query.

## DomainDataSource

### Loading Data

Running your application currently does very little. You should see an empty DataGrid and a number of TextBoxes and ComboBoxes. We will begin by loading data from your DomainContext into the application declaratively.

Make the highlighted additions to your UserControl in MainPage.xaml:

<Grid>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<riaControls:DomainDataSource x:Name="source" QueryName="GetEmployees" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

</riaControls:DomainDataSource>

<data:DataGrid ItemsSource="{Binding Data, ElementName=source}" />

<Grid Grid.Row="1">

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<data:DataPager Grid.Row="0" Grid.ColumnSpan="2" Source="{Binding Data, ElementName=source}" />

This creates a DomainDataSource and gives it access to an instance of your DomainContext. By specifying a QueryName, the DomainDataSource is able to manage loading the data when your application launches. By setting AutoLoad to “True” we allow the DomainDataSource to reload data whenever the query changes. Right now, our query is simple: “Load all of the data using the GetEmployees method.” As a result, the DomainDataSource will only load data when the application is launched.

Run the application to see the loaded data.

### Parameterized Query Methods

In our setup for the project, we added a parameterized method to our AdventureWorksDomainService (and hence a parameterized query method to the generated AdventureWorksDomainContext). This method filters the employees by their gender. To use this parameterized query method to get all Male employees, we make the highlighted changes to our code:

<riaControls:DomainDataSource x:Name="source" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:Parameter ParameterName="gender" Value="M" />

</riaControls:DomainDataSource.QueryParameters>

</riaControls:DomainDataSource>

When you run your application, you will see that the only employees loaded are ones with a Male gender.

Rather than specifying an unchanging value for the parameter as above, we may choose to allow our user to change its value. The DomainDataSource supports this using ControlParameters. We can bind this parameter to a ComboBox like so:

<riaControls:DomainDataSource x:Name="source" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:ControlParameter ParameterName="gender"

ControlName="gender"

PropertyName="SelectedItem.Content"

RefreshEventName="SelectionChanged" />

</riaControls:DomainDataSource.QueryParameters>

</riaControls:DomainDataSource>

In general, you can use ControlParameters with the DomainDataSource to link the query that the DomainDataSource is issuing to the DomainContext to your UI. If the DomainDataSource is set to AutoLoad, it will automatically load the updated query whenever those linked values are changed.

### Shaping Data

The DomainDataSource allows you to change the shape of the data retrieved using the DomainContext. Using the DomainDataSource, developers can declare how the data that the DomainContext loads should be sorted, grouped, filtered, and paginated. Furthermore, it will respect adjustments to the query made through the ICollectionView interface on the data that it exposes, allowing controls and developers to manipulate the query without relying on the DomainDataSource itself.

#### Sorting

The DomainDataSource makes sorting the data it loads simple. By adding SortDescriptors to the DomainDataSource a developer can specify the properties on the data that should be used for ordering.

We can sort our employees first by title, then by login id by modifying our XAML like so:

<riaControls:DomainDataSource x:Name="source" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:ControlParameter ParameterName="gender"

ControlName="gender"

PropertyName="SelectedItem.Content"

RefreshEventName="SelectionChanged" />

</riaControls:DomainDataSource.QueryParameters>

<riaControls:DomainDataSource.SortDescriptors>

<riaData:SortDescriptor PropertyPath="Title" Direction="Ascending" />

<riaData:SortDescriptor PropertyPath="LoginID" Direction="Ascending" />

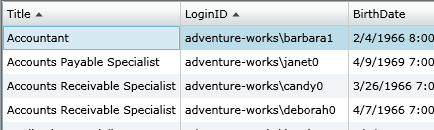
</riaControls:DomainDataSource.SortDescriptors>

</riaControls:DomainDataSource>

Here, we have added two SortDescriptors and set their PropertyPath to be the path on the entity that will be used for ordering. For more complex entity types, this PropertyPath supports dotted syntax (e.g. “Position.Title”). By setting the Direction on the SortDescriptor, we can choose whether the entities should be sorted in ascending or descending order.

Running the application, we can see that the data is sorted by Title and LoginID, as we would expect. If we change our “gender” parameter, the data will be reloaded, still in the order we requested.

Looking at the resulting UI, you’ll notice that the DataGrid actually reflects our SortDescriptors in its column headers:



This is possible because the Data property of the DomainDataSource is an ICollectionView. Working with this interface, the DataGrid is able to manipulate the data as well.

Try clicking on a few of the columns of the DataGrid (or Shift-Clicking to sort by multiple columns). You’ll notice that after a short delay (as the data is retrieved), the data in the grid updates appropriately. If you were handling LoadedData events from the DomainDataSource, you would see that clicking the column headers to sort the data in the DataGrid actually causes the DomainDataSource to update its query and reload its data. This clean integration through the ICollectionView interface makes it possible to create controls like the DataGrid that will dynamically query data through the DomainDataSource or other providers of ICollectionView implementations.

#### Grouping

Much like sorting, grouping data using the DomainDataSource can be accomplished declaratively. Controls like the DataGrid understand grouping using the ICollectionView interface, and are thus able to reflect groupings produced by the DomainDataSource transparently.

Suppose we wished to group by ManagerID to see all employees that share a manager together.

To accomplish this, we modify the XAML as follows:

<riaControls:DomainDataSource x:Name="source" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:ControlParameter ParameterName="gender"

ControlName="gender"

PropertyName="SelectedItem.Content"

RefreshEventName="SelectionChanged" />

</riaControls:DomainDataSource.QueryParameters>

<riaControls:DomainDataSource.SortDescriptors>

<riaData:SortDescriptor PropertyPath="Title" Direction="Ascending" />

<riaData:SortDescriptor PropertyPath="LoginID" Direction="Ascending" />

</riaControls:DomainDataSource.SortDescriptors>

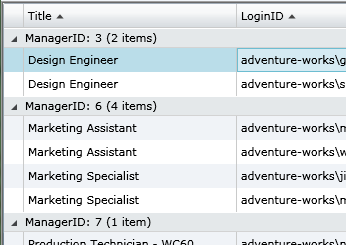
<riaControls:DomainDataSource.GroupDescriptors>

<riaData:GroupDescriptor PropertyPath="ManagerID" />

</riaControls:DomainDataSource.GroupDescriptors>

</riaControls:DomainDataSource>

Running the application, we see that the data has been grouped by the entities’ ManagerIDs:



If it is prudent to do so, we can add additional GroupDescriptors to the DomainDataSource. This would allow us to, for example, group locations first by country, then by city.

#### Filtering

The DomainDataSource also supports adding filters to the query. By adding FilterDescriptors, we can declaratively choose which entities should be loaded from the DomainContext. FilterDescriptors support a number of operations on strings and types that support comparison. Furthermore, FilterDescriptors can be driven by ControlParameters, allowing the user to easily manipulate the query.

We have already prepared a number of TextBoxes that will allow the user to modify the filters on the DomainDataSource. We can create the appropriate filter descriptors by adding the following to our DomainDataSource:

<riaControls:DomainDataSource x:Name="source" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:ControlParameter ParameterName="gender"

ControlName="gender"

PropertyName="SelectedItem.Content"

RefreshEventName="SelectionChanged" />

</riaControls:DomainDataSource.QueryParameters>

<riaControls:DomainDataSource.SortDescriptors>

<riaData:SortDescriptor PropertyPath="Title" Direction="Ascending" />

<riaData:SortDescriptor PropertyPath="LoginID" Direction="Ascending" />

</riaControls:DomainDataSource.SortDescriptors>

<riaControls:DomainDataSource.GroupDescriptors>

<riaData:GroupDescriptor PropertyPath="ManagerID" />

</riaControls:DomainDataSource.GroupDescriptors>

<riaControls:DomainDataSource.FilterDescriptors>

<riaData:FilterDescriptorCollection LogicalOperator="And">

<riaData:FilterDescriptor PropertyPath="LoginID" Operator="Contains">

<riaData:ControlParameter ControlName="loginID" PropertyName="Text" RefreshEventName="TextChanged"/>

</riaData:FilterDescriptor>

<riaData:FilterDescriptor PropertyPath="ContactID" Operator="IsEqualTo">

<riaData:ControlParameter ControlName="contactID" PropertyName="Text" RefreshEventName="TextChanged" />

</riaData:FilterDescriptor>

<riaData:FilterDescriptor PropertyPath="VacationHours" Operator="IsGreaterThanOrEqualTo">

<riaData:ControlParameter ControlName="vacationHours" PropertyName="Text" RefreshEventName="TextChanged" />

</riaData:FilterDescriptor>

</riaData:FilterDescriptorCollection>

</riaControls:DomainDataSource.FilterDescriptors>

</riaControls:DomainDataSource>

<data:DataGrid ItemsSource="{Binding Data, ElementName=source}" />

Here we have added three FilterDescriptors which will update the query based on user input. Multiple filter descriptors can either be applied with a logical operation of “And” or “Or”, which will query for items that match all of the filters or items that match any of the filters, respectively.

Running the application, you can filter the data by entering text into the textboxes below the DataGrid. Note that each textbox filters using a different operation. LoginID filters entities that contain the entered text, ContactID filters entities whose contact ID equals the entered number, and Vacation hours filters entities that have more vacation hours than the entered number.

#### Paging

The DomainDataSource also improves the experience working with large amounts of data. By setting a PageSize and LoadSize, the DomainDataSource will limit the number of entities downloaded at a time by dividing the entire query result into LoadSize-sized pieces for download, and displaying them for the user in PageSize-sized “pages.” More data will only be requested when the user attempts to move to a page that has not already been loaded. The result is a series of smaller requests for data and a paginated UI.

To interact with pages, we use the DataPager control (bound to the Data on the DomainDataSource in the “Loading Data” step). This control gives users UI for moving from page to page in the data, and will trigger load requests when the data is unavailable.

To add paging and a LoadSize to the DomainDataSource, make the following changes to your XAML:

<riaControls:DomainDataSource x:Name="source" PageSize="15" LoadSize="30" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:ControlParameter ParameterName="gender"

ControlName="gender"

PropertyName="SelectedItem.Content"

RefreshEventName="SelectionChanged" />

</riaControls:DomainDataSource.QueryParameters>

<riaControls:DomainDataSource.SortDescriptors>

<riaData:SortDescriptor PropertyPath="Title" Direction="Ascending" />

<riaData:SortDescriptor PropertyPath="LoginID" Direction="Ascending" />

</riaControls:DomainDataSource.SortDescriptors>

<riaControls:DomainDataSource.GroupDescriptors>

<riaData:GroupDescriptor PropertyPath="ManagerID" />

</riaControls:DomainDataSource.GroupDescriptors>

<riaControls:DomainDataSource.FilterDescriptors>

<riaData:FilterDescriptorCollection LogicalOperator="And">

<riaData:FilterDescriptor PropertyPath="LoginID" Operator="Contains">

<riaData:ControlParameter ControlName="loginID" PropertyName="Text" RefreshEventName="TextChanged"/>

</riaData:FilterDescriptor>

<riaData:FilterDescriptor PropertyPath="ContactID" Operator="IsEqualTo">

<riaData:ControlParameter ControlName="contactID" PropertyName="Text" RefreshEventName="TextChanged" />

</riaData:FilterDescriptor>

<riaData:FilterDescriptor PropertyPath="VacationHours" Operator="IsGreaterThanOrEqualTo">

<riaData:ControlParameter ControlName="vacationHours" PropertyName="Text" RefreshEventName="TextChanged" />

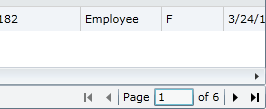
</riaData:FilterDescriptor>

</riaData:FilterDescriptorCollection>

</riaControls:DomainDataSource.FilterDescriptors>

</riaControls:DomainDataSource>

When you run the application, you will notice that the DataPager is now enabled, allowing you to page through the data.



### Editing

The DomainDataSource supports modifications to the data it retrieves through the IEditableCollectionView interface on the Data property and directly through the entities. It will respect changes to the underlying entity lists on its DomainContext, and has API for submitting changes to the underlying DomainContext.

Specifically, the DomainDataSource has a SubmitChanges() method and a RejectChanges() method, which submit and cancel changes, respectively, to all data in the DomainContext.

## Summary

The DomainDataSource is an interface to the DomainContext that exposes data loading and queries to the UI and allows developers to specify declaratively (in XAML) how data should be retrieved, shaped, and presented. With the DomainDataSource, it is easy to create RIA Silverlight applications with a rich data experience.

## See Also

* Understanding N-Tier Silverlight Application Projects

## Reference XAML

After following all of the steps in this section, the XAML for the main page should be similar to the following XAML:

<UserControl x:Class="<YourSilverlightApplicationNamespace>.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:d="http://schemas.microsoft.com/expression/blend/2008" xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"

xmlns:riaControls="clr-namespace:System.Windows.Controls;assembly=System.Windows.Ria.Controls"

xmlns:riaData="clr-namespace:System.Windows.Data;assembly=System.Windows.Ria.Controls"

xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"

xmlns:domain="clr-namespace:<YourSilverlightApplicationNamespace>.Web"

mc:Ignorable="d" d:DesignWidth="640" d:DesignHeight="480">

<Grid>

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition Height="Auto" />

</Grid.RowDefinitions>

<riaControls:DomainDataSource x:Name="source" PageSize="15" LoadSize="30" QueryName="GetEmployeesByGender" AutoLoad="True">

<riaControls:DomainDataSource.DomainContext>

<domain:AdventureWorksDomainContext />

</riaControls:DomainDataSource.DomainContext>

<riaControls:DomainDataSource.QueryParameters>

<riaData:ControlParameter ParameterName="gender"

ControlName="gender"

PropertyName="SelectedItem.Content"

RefreshEventName="SelectionChanged" />

</riaControls:DomainDataSource.QueryParameters>

<riaControls:DomainDataSource.SortDescriptors>

<riaData:SortDescriptor PropertyPath="Title" Direction="Ascending" />

<riaData:SortDescriptor PropertyPath="LoginID" Direction="Ascending" />

</riaControls:DomainDataSource.SortDescriptors>

<riaControls:DomainDataSource.GroupDescriptors>

<riaData:GroupDescriptor PropertyPath="ManagerID" />

</riaControls:DomainDataSource.GroupDescriptors>

<riaControls:DomainDataSource.FilterDescriptors>

<riaData:FilterDescriptorCollection LogicalOperator="And">

<riaData:FilterDescriptor PropertyPath="LoginID" Operator="Contains">

<riaData:ControlParameter ControlName="loginID" PropertyName="Text" RefreshEventName="TextChanged"/>

</riaData:FilterDescriptor>

<riaData:FilterDescriptor PropertyPath="ContactID" Operator="IsEqualTo">

<riaData:ControlParameter ControlName="contactID" PropertyName="Text" RefreshEventName="TextChanged" />

</riaData:FilterDescriptor>

<riaData:FilterDescriptor PropertyPath="VacationHours" Operator="IsGreaterThanOrEqualTo">

<riaData:ControlParameter ControlName="vacationHours" PropertyName="Text" RefreshEventName="TextChanged" />

</riaData:FilterDescriptor>

</riaData:FilterDescriptorCollection>

</riaControls:DomainDataSource.FilterDescriptors>

</riaControls:DomainDataSource>

<data:DataGrid ItemsSource="{Binding Data, ElementName=source}" />

<Grid Grid.Row="1">

<Grid.RowDefinitions>

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

<RowDefinition />

</Grid.RowDefinitions>

<Grid.ColumnDefinitions>

<ColumnDefinition Width="Auto" />

<ColumnDefinition />

</Grid.ColumnDefinitions>

<data:DataPager Grid.Row="0" Grid.ColumnSpan="2" Source="{Binding Data, ElementName=source}" />

<TextBlock Grid.Row="1" Text="LoginID" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBlock Grid.Row="2" Text="ContactID" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBlock Grid.Row="3" Text="Gender" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBlock Grid.Row="4" Text="Minimum Vacation Hours" HorizontalAlignment="Right" VerticalAlignment="Center" Margin="4" />

<TextBox x:Name="loginID" Grid.Row="1" Grid.Column="1" Margin="4" />

<TextBox x:Name="contactID" Grid.Row="2" Grid.Column="1" Margin="4" />

<ComboBox x:Name="gender" Grid.Row="3" Grid.Column="1" Margin="4" SelectedIndex="1">

<ComboBoxItem Content="M" />

<ComboBoxItem Content="F" />

</ComboBox>

<TextBox x:Name="vacationHours" Grid.Row="4" Grid.Column="1" Margin="4" />

</Grid>

</Grid>

</UserControl>

# How to use metadata classes

.NET RIA Services supports the use of additional metadata classes that can be used to adorn existing partial types with additional metadata. This is useful in situations where types are defined in generated code and cannot be safely modified by hand.

For example, let’s assume a user is using the ADO.NET Entity Framework or LINQ-to-SQL to model and generate entities that will be exposed by a DomainService. In this scenario, there is no reasonable way for a developer to annotate existing properties with additional metadata without risk of losing changes when those entity types are regenerated.

Using a metadata class enables a developer to maintain additional metadata in a separate class.

## Procedure

To create and use a metadata class

### Prerequisites:

* Your project already contains entity types generated from a source such as ADO.NET Entity Framework.
* The types that will be extended with metadata classes are declared as partial types.

### Steps (using Wizard):

1. In your project, add a new item and select “Domain Service Class” as the type. This will start the domain service wizard.

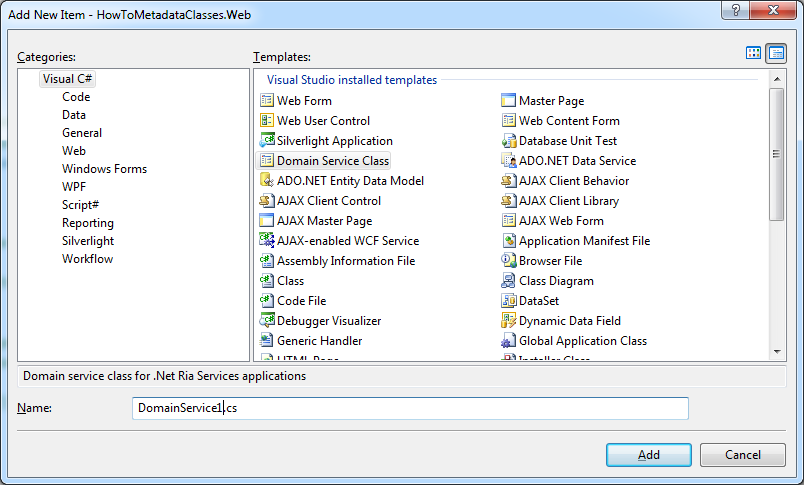


Figure - the Visual Studio "Add New Item" dialog with the Domain Service class highlighted.

1. Check the box at bottom to “Generate associated classes for metadata”.

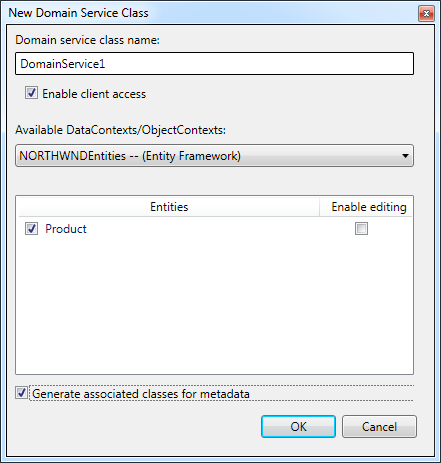


Figure - the Domain Service wizard dialog.

1. Your project should have 2 new files: one generated DomainService and an accompanying DomainService metadata class. (“<name>.metadata.cs”)

### Steps (manual):

1. In your project, add a new class that will be used to extend an existing entity with metadata.
2. Add fields to your metadata class that map to the targeted existing entity properties by using the same property types and names.

**Note:** Metadata class associations are case-sensitive.

1. Within the same metadata class, add an additional partial class declaration of the entity type that you want to extend. (Be sure to define this in the same namespace as that of the original entity.)
2. Annotate the partial class declaration with a MetadataTypeAttribute, passing a type reference to your metadata type in the attribute constructor. (This provides a hint to the underlying framework that the metadata class should be examined for additional information.)

Now you can add additional attributes to the metadata class fields. This additional metadata will be treated as if it resided directly on the entity types during framework validation and client code generation.

## Example

In this example, we’ll create a metadata for an existing type named “Product”. The **Product** class was generated by ADO.NET Entity Framework and exists as a *partial* type declaration.

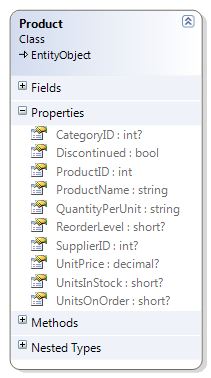


Figure - an example Product entity type, generated from the Northwind database.

1. Create new class in your server assembly named “ProductMetadata.cs”
2. Within this new code file, create a class definition for **ProductMetadata**.
3. Also create a partial definition for the **Product** class—making sure that this partial definition resides in the same namespace as the generated entity type.
4. Add a **MetadataType** attribute to the **Product** partial type defined in the step above, passing a type reference to **ProductMetadata** as the constructor argument. With this in place, the RIA Services infrastructure will know to examine the **ProductMetadata** type for additional **Product** metadata.

With all the necessary pieces in place, we can begin to describe additional **Product** metadata within the **ProductMetadata** class. In this example, we’ll provide additional metadata information used to describe and validate the **ProductName** property.

1. Add a field of type string to **ProductMetadata** called **ProductName**. Any metadata annotations you add here will be projected onto the entity’s property.

|  |
| --- |
| namespace NorthwindExample  {  using System;  using System.ComponentModel.DataAnnotations;  using System.Data;  // The MetadataTypeAttribute allows us to associated the  // ProductMetadata type with our Product entity type.  [MetadataType(typeof(ProductMetadata))]  public partial class Product { }  // This class is used to provide additional metadata.  public partial class ProductMetadata  {  // Adding additional validation metadata to the  // Product.ProductName property by placing it here.  [Required]  [RegularExpression("[A-Z][A-Za-z0-9]\*")]  [StringLength(32)]  public string ProductName;  }  } |

With this metadata class association in place, RIA Services will respect and use this additional metadata during validation and client code generation.

## External Metadata

As an alternative to using metadata classes, metadata information can also be declared and loaded from an external source such as XML. See the “[Using External Metadata](http://code.msdn.microsoft.com/RIAServices)” sample.

## See Also

* How to add Validation to Entities
* Understanding N-Tier Silverlight Application Projects
* Understanding Silverlight Client Code Generation
* Using External Metadata

# How to Add Validation to Entities

Silverlight and .NET RIA Services offer a robust validation framework that can be used to enforce validation constraints on entities and DomainService operations. Validation logic can be added using custom attributes such as those found in the System.ComponentModel.DataAnnotations namespace:

* CustomValidationAttribute
* DataTypeAttribute
* RangeAttribute
* RegularExpressionAttribute
* RequiredAttribute
* StringLengthAttribute

Validation attributes on server-side entities exposed by a domain service are propagated to their generated client entity representations. The generated entity property set-methods will evaluate validation attributes at runtime and ensure the validation rules are upheld.

## Procedure

### Prerequisites:

* You have at least 1 domain service with client access enabled.
* You have at least 1 entity exposed by a domain service.

### Steps:

1. Decorate an entity type or entity type properties with validation attributes as appropriate.

* Consider using a Metadata class to define validation attributes on code-generated entity types.

1. Build your solution. Validation attributes should be reflected on the generated client.

### Notes:

* If attempting to add validation attributes to an entity type generated from tools such as ADO.NET Entity Framework, use a metadata class to define validation attributes.
* If using the CustomValidationAttribute on types or properties that are exposed on the client, these attributes will only be propagated over to the generated client types if the client project also defines the validator type and method specified in the CustomValidationAttribute constructor. (See “How to Share Code Across Tiers”).

## Example

In this example, we’ll build a simple Restaurant Locator service. The goal of this service will be to allow users to add, update and query restaurant information.

To start, let’s assume we have a very simple pre-populated database that contains information about Restaurants.

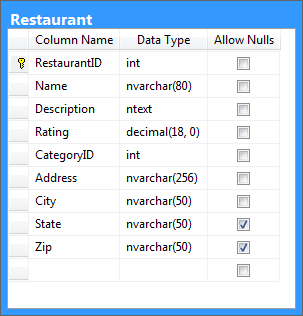


Figure - simple database representation of a restaurant.

First, we create an Entity Framework data model and select create a server entity to represent restaurants. We can do this by adding a new item to the server project and selecting “ADO.NET Entity DataModel” as our item type. Then proceed through the wizard, selecting the Restaurant table from our sample database.

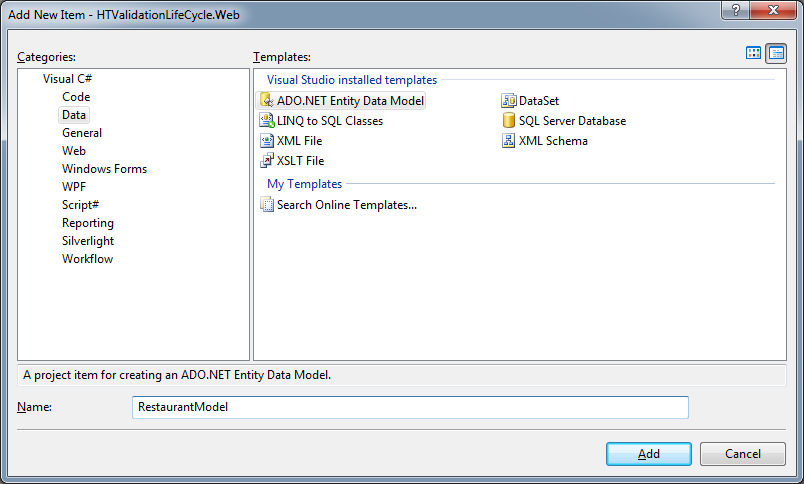


Figure - Visual Studio "Add New Item" dialog showing the ADO.NET Entity Data Model item.

After walking through the wizard, we should have an auto-generated Restaurant type in our project. We will use this entity later in our Domain Service.

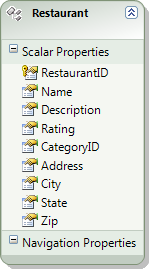


Figure - ADO.NET Entity Framework entity generated from database table.

Next, we’ll add a domain service called “RestaurantLocatorService”. **Be sure to compile your solution at this point so that all of the Entity Framework types are generated.**

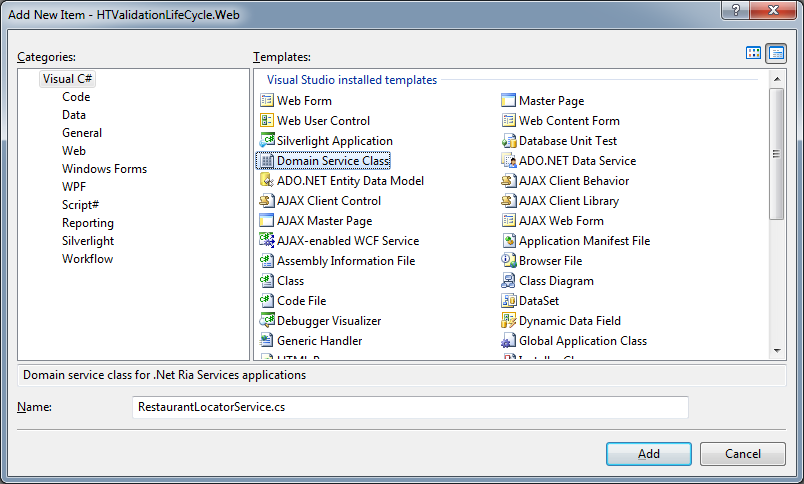


Figure - Visual Studio "Add New Item" dialog showing the Domain Service item.

By adding a new Domain Service item, the Domain Service wizard will start and allow us to delete the entities we wish to expose in our service. Check the “Generate associated classes for metadata” option.

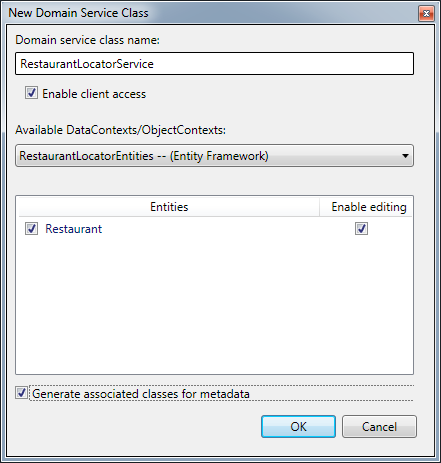


Figure - The Domain Service wizard.

This will add 2 new files to our solution: RestaurantLocatorService.cs and RestaurantLocatorService.metadata.cs. (Or \*.vb in the case of Visual Basic.)

Now before we begin adding our metadata information, we’ll create 1 more *shared* class by adding a new file called “RestaurantValidator.shared.cs”. The “.shared.cs” (or “.shared.vb”) is recognized by the project system and will be copied over to the client project during builds. This allows us to perform common validations on client and server.

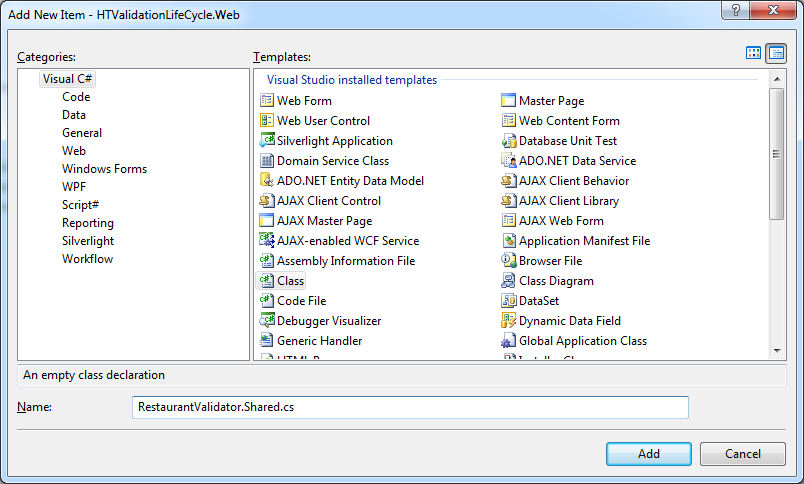


Figure - Adding a shared source file.

By naming our source file \*.Shared.\*, the underlying project infrastructure will automatically copy this file into the client application when we build. This allows us to easily maintain a shared code file in one location without the need to create Visual Studio linked file references.

In RestaurantValidator, we define 1 static method called “IsRestaurantValid” that will be invoked later to validate Restaurant instances. In this example, we will fake verification of the city, state and zip code combination.

|  |
| --- |
| namespace RestaurantLocator  {  using System;  using System.ComponentModel.DataAnnotations;  using System.Web.Ria.Data;  public class RestaurantValidator  {  public static ValidationResult IsRestaurantValid(  Restaurant restaurant,  ValidationContext context)  {  bool valid = IsAddressValid(  restaurant.City,  restaurant.State,  restaurant.Zip);  if (!valid)  {  return new ValidationResult(  "Invalid location information. Please check values.");  }  return null;  }  }  } |

Using the generated “RestaurantLocatorService.metadata” metadata class, we can begin adding validation metadata to our Restaurant entity. (Added validation attributes are highlighted.)

|  |
| --- |
| namespace RestaurantLocator  {  using System;  using System.ComponentModel.DataAnnotations;  [CustomValidation(typeof(RestaurantValidator), "IsRestaurantValid")]  [MetadataTypeAttribute(typeof(Restaurant.RestaurantMetadata))]  public partial class Restaurant  {  internal sealed class RestaurantMetadata  {  [Required]  [StringLength(80, MinimumLength = 2)]  public string Name;  [Required]  [StringLength(2048, MinimumLength = 10)]  public string Description;  [Range(0, 10)]  public int Rating;  [Required]  public string Address;  [Required]  [StringLength(80, MinimumLength = 2)]  public string City;  [Required]  [StringLength(14, MinimumLength = 4)]  public string State;  [Required]  [RegularExpression(@"\d{5}|\d{5}-\d{4}")]  public string Zip;  }  }  } |

Now if we build the project, we can examine the generated client code and verify that our metadata has propagated across.

In the client Silverlight application, we can begin to make use of these new entities and related validation metadata by creating a DataGrid and corresponding DomainDataSource used to populate it.

|  |
| --- |
| <UserControl  x:Class="RestaurantLocator.MainPage"  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"  xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"  xmlns:ria="clr-namespace:System.Windows.Controls;assembly=System.Windows.Ria.Controls"  xmlns:services="clr-namespace:RestaurantLocator.Services"  xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data">  <Grid x:Name="LayoutRoot" Background="White">  <ria:DomainDataSource  x:Name="RestaurantDataSource"  QueryName="LoadRestaurants">  <ria:DomainDataSource.DomainContext>  <services:RestaurantLocatorContext />  </ria:DomainDataSource.DomainContext>  </ria:DomainDataSource>  <data:DataGrid  x:Name="RestaurantDataGrid"  ItemsSource="{Binding ElementName=RestaurantDataSource, Path=Data}" />  </Grid>  </UserControl> |

Now if we run this application, we can see our Restaurant data

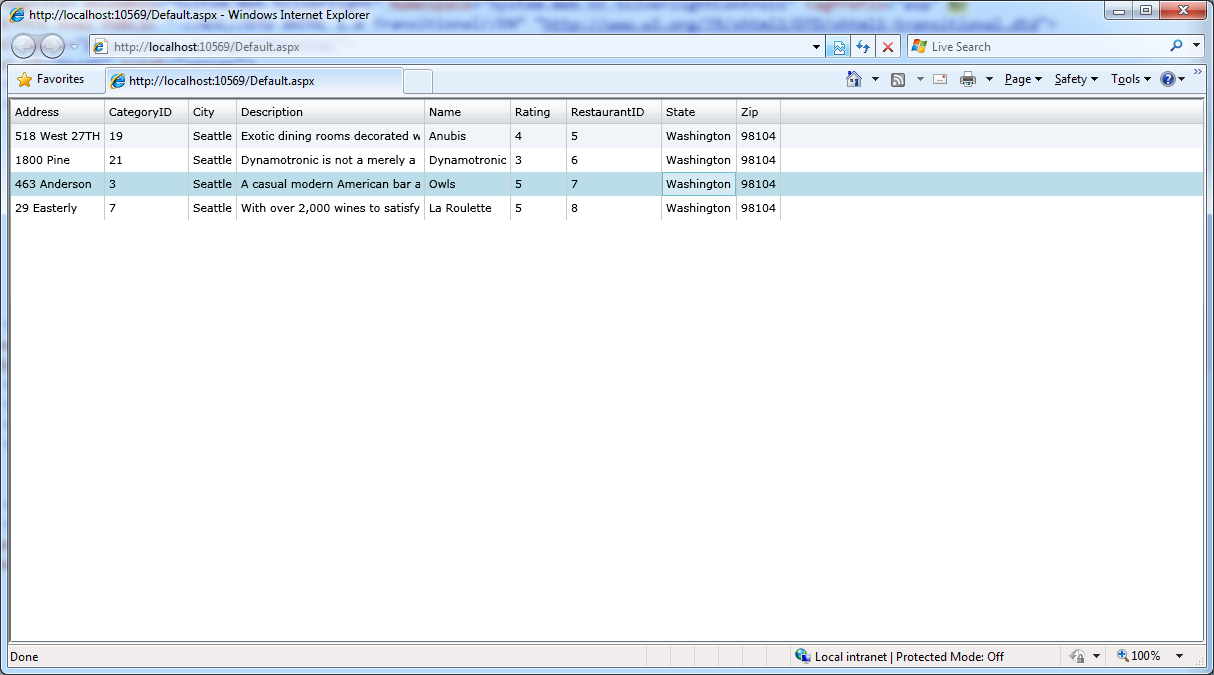


Figure - Sample restaurant data displayed in a data grid.

To see validation in action, we can try to edit a row and enter incorrect data. For example, if we try to make the “Description” value null we see the following error:

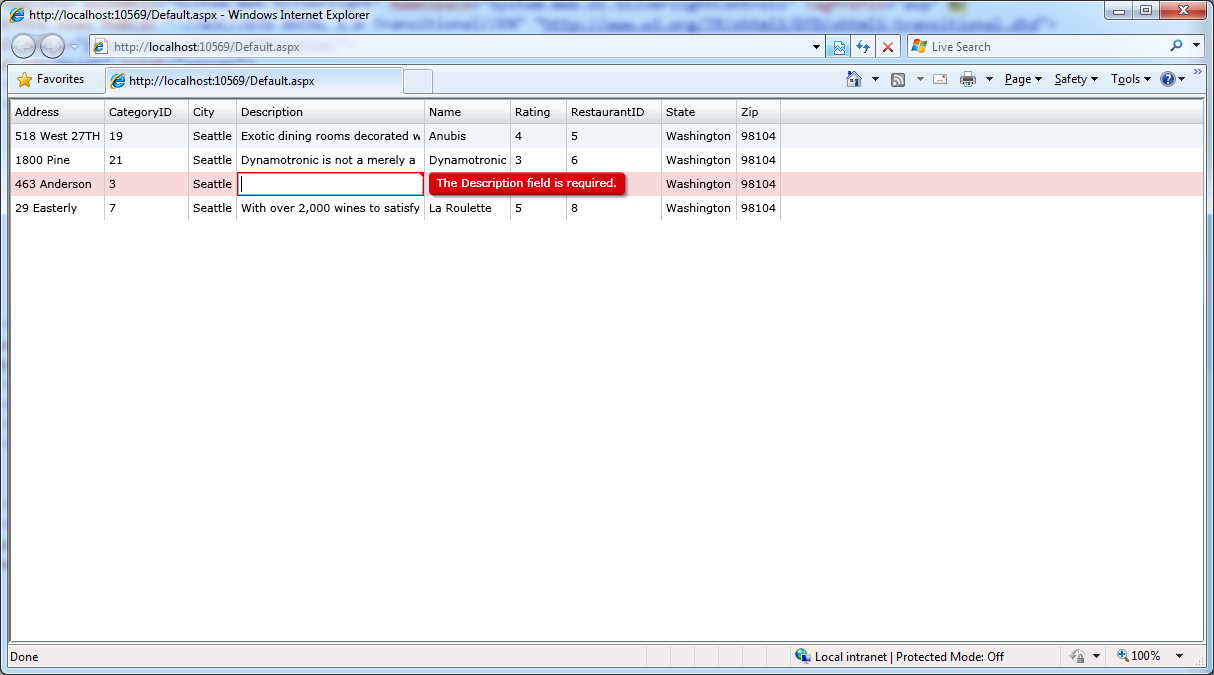


Figure - Example validation error.

Property-level validation happens at the time of setting an entity property. Entity-level validation (such as the shared RestaurantValidator we defined earlier) happens *after* property validation.

You can also work with validation in code as shown in the client-side example below.

|  |
| --- |
| public void UpdateRestaurant(Restaurant restaurant)  {  try  {  // The following will raise a ValidationException due to the  // RequiredAttribute validation attributes applied in the  // "RestraurantMetadata" metadata class defined above.  restaurant.Name = null;  }  catch (ValidationException ve)  {  // Display error to user  // ...  }  } |

## Extensibility

One can also create their own validation attributes by deriving from the System.ComponentModel.DataAnnotations.ValidationAttribute and implementing custom logic in the IsValid methods.

## See Also

* Understanding N-tier Silverlight Application Projects
* How to use metadata classes
* How to Share Code Across Tiers

# How to Share Code Across Tiers

One advantage of using ASP.NET on the mid-tier and Silverlight on the client tier is that both share the same Common Language Runtime (CLR). Business logic written in a language like C# or Visual Basic can run on both tiers.

This section describes how to use .NET RIA Services to write business logic once and use it on both tiers.

For the sake of this discussion, there are 3 different kinds of shared code:

1. Generated proxy types
2. Shared source files
3. Shared assemblies

## Generated proxy types

One kind of code sharing happens automatically whenever you create an application using .NET RIA Services. It is described here to help you understand and take advantage of this built-in plumbing.

When the “Enable .NET RIA Services” option is checked for a Silverlight project, the build process is extended to generate proxy types in that project based on the types found in the corresponding mid-tier project[[13]](#footnote-13). This proxy type generation is described in the section “” and can be summarized as:

* For every DomainService type on the mid-tier marked with the EnableClientAccessAttribute, generate a DomainContext type on the client tier.
* For every entity type exposed by the DomainService[[14]](#footnote-14), generate a corresponding proxy entity type on the client tier.
* For every public property exposed by the entity type, generate a corresponding property in the proxy entity type.
* For every custom attribute on the entity type or its properties (such as RequiredAttribute), duplicate that custom attribute in the generated proxy entity type.

Key:

Silverlight project

Web project

**RIA Link**

**MyDomainService.cs**

**MyDomainContext.cs**

*Proxy types are generated into the client project when it is built*

**MyEntity.cs**

**Developer-written**

**Automatically generated**

You can then write code that refers to these common entity types and be assured it will compile and run on both tiers. Moreover, any custom attributes appearing on the mid-tier such as RequiredAttribute or RangeAttribute will be propagated to the proxy types on the client. In this way, the validation rules declared on the mid-tier will also be applied by the property setters on the client.

It may help to think of this as *implicit* code sharing. If you use .NET RIA Services to build the application, create DomainService types on the mid-tier and create a “RIA Link” between the client and mid-tier projects, your client tier will automatically have equivalent proxy types and associated validation logic.

The other techniques described below require explicit actions by the developer.

## Shared source files

Another way to share code across tiers is by explicitly sharing source files. The code is written once and then compiled separately into each tier’s project(s). There are 2 ways to do this; by using common file naming patterns and by file links.

.NET RIA Services supports both styles, and you can use either or both within a solution. Each style has its own unique advantages described below.

### Sharing files by naming pattern

.NET RIA Services supports shared source files between projects though the use of a common naming pattern. All files in the mid-tier project named with the pattern \*.shared.cs (C#) or \*.shared.vb (Visual Basic) will be copied at build time to the client-tier project if there is a “RIA Link[[15]](#footnote-15)” between those projects.

It may help to think of this as a “push” model, because the shared files are actively copied from the mid-tier project to the client-tier project during compilation.

Web project

Silverlight project

**RIA Link**

**MyCode.shared.cs**

**MyCode.shared.cs**

*\*.shared.\* files are automatically copied into the client project during build*

Advantages:

***Support is built-in*** -- no special user interaction or hidden metadata is required to share files. As soon as they are created with this naming pattern, the client projects will have them the next time they are built.

***Transparent*** – even outside of Visual Studio, it is obvious which files are shared.

***Self-maintaining*** – as shared files are added to the mid-tier project, all client tier projects automatically pick them up the next time they are built. The client projects are not modified – this is strictly a compile-time operation.

Disadvantages:

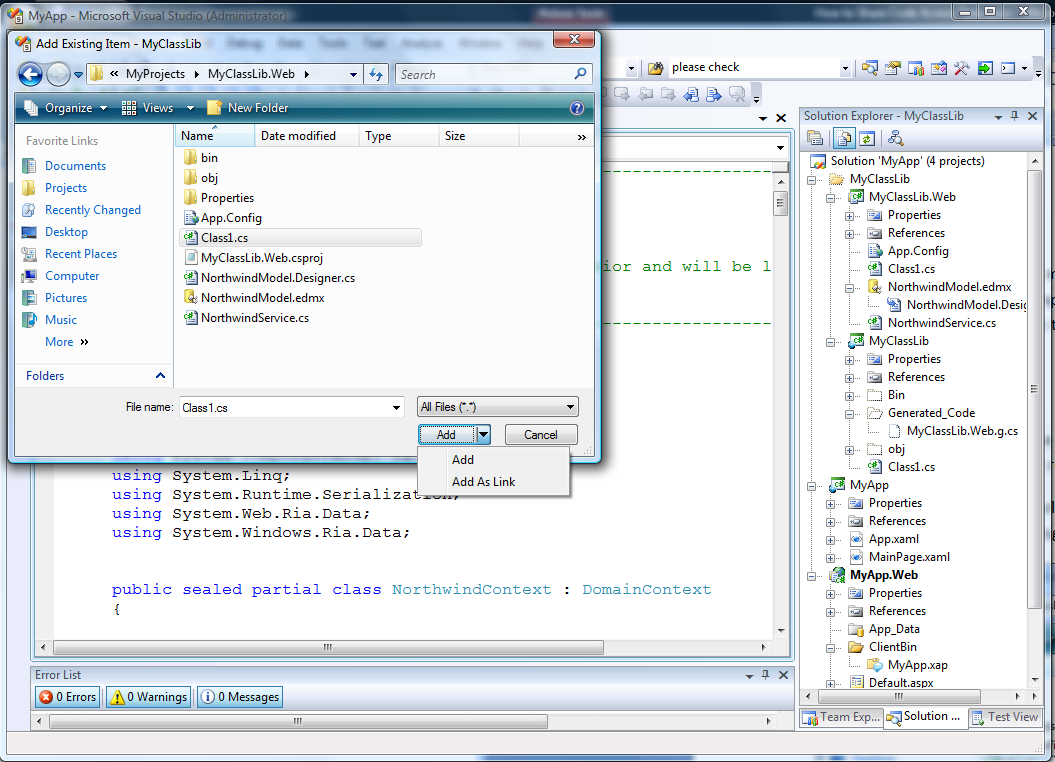
***New concept*** – developers need to be aware of this new mechanism.

***Files are copied*** – to make the debugging experience better[[16]](#footnote-16), the files are physically copied. It can be tempting to edit the copy by mistake.

### Sharing files by linking

The other common way to share source files between projects is via *linked files*. This mechanism is built into Visual Studio and does not require .NET RIA Services or the “RIA Link.”

To use it, right click on the Silverlight project, choose “Add Existing Item…,” select a file from the server project, and choose the “Add As Link” option in the dialog:



This adds a reference to the specified file into the set of files the project will compile. It does not create a copy of the file.

It may help to think of this as a “pull” model in that the client project merely refers back to the shared file in the mid-tier project.

Logically, it looks like this:

Web project

Silverlight project

**RIA Link**

**MyCode.cs**

**MyCode.cs**

*The client project simply refers to the original file in the mid-tier project*

Advantages:

***Uses existing VS mechanisms*** – there is no new concept to learn.

***Files are not copied*** – the user will not be tempted to edit the copy by mistake

Disadvantages:

**Requires explicit user action** – each shared file must be added to each client project through an explicit user interaction.

***Higher maintenance*** – as shared files are added to (or removed from) the mid-tier project, each client project must be updated.

***Not transparent*** – without examining the project file itself, it is not obvious which files are shared.

***Awkward debugging experience*** – because the same code runs on multiple tiers and in multiple apps, and because there is only a single physical file, setting breakpoints and single-stepping through the code can be confusing. Which tier or app hit the breakpoint?

The file linking mechanism does not depend on the RIA link or on any special relationship between the projects. For example, both the mid-tier and client-tier projects could use file links to refer to the same physical files in some other arbitrary project or folder on disk:

Web project

Silverlight project

**RIA Link**

**MyCode.cs**

*Client and mid-tier projects both refer to the same file in some other project*

Misc. project

**MyCode.cs**

**MyCode.cs**

## Shared class libraries

Another way to share code is class libraries. Rather than sharing source files between projects, the code is compiled into a class library, and then multiple projects share it at the binary level through assembly references.

Special challenges arise in sharing at the binary level between the mid-tier and the client-tier because they are compiled against different frameworks (.NET 3.5 SP1 for the mid-tier, and Silverlight v3.0 for the client tier). Assemblies are not currently binary compatible across these frameworks.

This can be solved using .NET RIA Services class libraries. This mechanism allows a .NET 3.5 class library and a Silverlight v3.0 class library to be treated as a single logical component. These libraries can use the other techniques described above to share common source files between them, and once they are compiled, the mid-tier and client-tier can just take binary dependencies on the generated libraries.

This diagram shows how an N-tier application could be organized using .NET RIA Services class libraries. In this case, both the mid-tier and client tier projects share code at the binary level by using conventional assembly references to their respective class libraries:

**.NET RIA Services Class Lib**

.NET RIA Services App

.NET Class Lib2

Silverlight App1

Web App

**RIA Link**

**Shared code2**

**DomainService2**

**.NET RIA Services Class Lib**

.NET Class Lib1

**RIA Link**

**Shared code1**

**DomainService1**

SL Class Lib1

Silverlight App2

**Shared code 1**

**DomainContext1**

SL Class Lib2

**Shared code 2**

**DomainContext2**

This subject is described in greater detail in the “” section.

## Summary

Because ASP.NET and Silverlight share the Common Language Runtime, business logic can be written once and run on both the mid-tier and the client-tier. .NET RIA Services provides built-in mechanisms to create entity proxy types on the client and to share code between the tiers either through shared source files or common class libraries.

# Using Authentication, Roles and Profiles

The Authentication Service provided in the RIA Services framework encapsulates authentication, authorization, and user profile. It makes verifying and tracking user identity a simple part of application development. This section will outline how to setup authentication, roles, and profile in a few simple steps.

## Authentication

### Server Tasks

#### Configure ASP.NET

Authentication is closely tied to the Membership framework provided by ASP.NET. The first step to enabling authentication is to correctly configure the *web.config* file in the Web project.

#### Windows Authentication

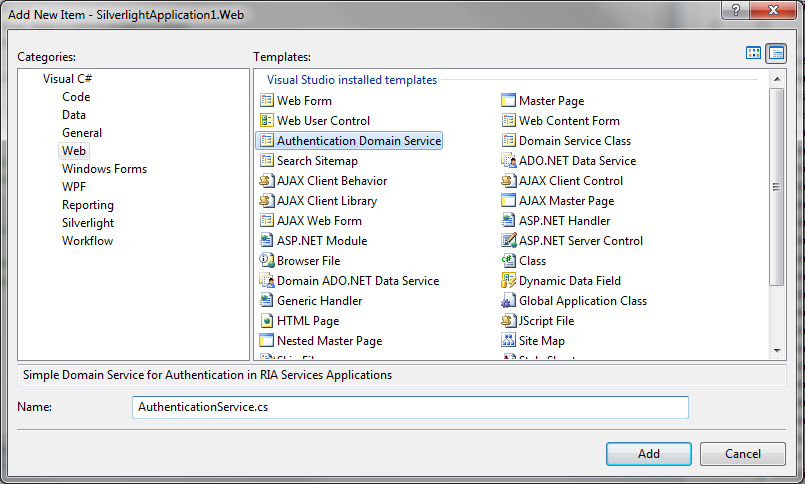
The configuration section *<system.web.authentication>* is set to *“Windows”* mode by default. Without any need for modification, your website will support Windows authentication out of the box.

#### Forms Authentication

To enable forms authentication, set the *<system.web.authentication>* configuration section mode to *“Forms”*. By default, forms authentication will be validated against the user table in a local database. For more information on how to configure and manage the database, the first few minutes of this ASP.NET [video](http://www.asp.net/learn/3.5-videos/video-45.aspx) provide a great introduction. Also, the ASP.NET [wiki](http://wiki.asp.net/page.aspx/43/authentication-and-authorization/) provides guidance for this and other security-related tasks.

#### Create a DomainService

The authentication state needs to be available on the client so the next step is setup a service endpoint using the Domain Service framework. For simplicity we’ll use the Authentication Domain Service item template in Visual Studio by opening the popup menu in the Web project and selecting “Add -> New Item…”



The template uses two classes provided by the framework, *AuthenticationBase* and *UserBase*. Together, these implement the basic contract for an Authentication Domain Service. The following snippets are abbreviated versions of the classes created by the template.

##### Visual Basic

<EnableClientAccess()> \_

Public Class AuthenticationService

Inherits AuthenticationBase(Of User)

End Class

Public Class User

Inherits UserBase

End Class

##### C#

[EnableClientAccess]

public class AuthenticationService : AuthenticationBase<User> { }

public class User : UserBase { }

### Client Tasks

#### Add the RiaContext to the Application

The *RiaContext* class makes accessing the Authentication Service and User convenient in code and XAML. It will be automatically generated for you in a RIA Services application. To make the *RiaContext* available we will need to add an instance to the application lifetime object collection in our application.

<Application xmlns:app="SampleApplication" ...>

...

<Application.ApplicationLifetimeObjects>

<app:RiaContext/>

</Application.ApplicationLifetimeObjects>

</Application>

#### Add an AuthenticationService to RiaContext

To enable authentication, we simply need to add a *WindowsAuthentication* or *FormsAuthentication* Service to the *RiaContext*. The type of the service should correspond to the authentication mode selected in *web.config* file in the Web project.

<Application xmlns:app="clr-namespace:SampleApplication"

xmlns:appsvc="clr-namespace: System.Windows.Ria.ApplicationServices;assembly=System.Windows.Ria" ...

>

...

<Application.ApplicationLifetimeObjects>

<app:RiaContext>

<app:RiaContext.Authentication>

<appsvc:WindowsAuthentication/>

<!-- Optionally <appsvc:FormsAuthentication/> -->

</app:RiaContext.Authentication>

</app:RiaContext>

</Application.ApplicationLifetimeObjects>

</Application>

### Client Usage

After completing the preceding steps, the Authentication Service will be fully functional. Using the service in code or XAML generally involves interacting with the generated *RiaContext* class. The Authentication Service instance you just created in XAML is available from *RiaContext.Current.Authentication*. Also, the state of the user is always available at *RiaContext.Current.User*. The *AuthenticationService.Login* and *AuthenticationService.Logout* methods are provided for Forms authentication, and *AuthenticationService.LoadUser* can be used to refresh the state of the user at any time.

#### Make the RiaContext available in XAML

We need to add the current *RiaContext* to our application resources to make it available to XAML. This should be done in the *Application.Startup* event before creating the root visual.

##### Visual Basic

Private Sub Application\_Startup(ByVal o As Object, ByVal e As StartupEventArgs) Handles Me.Startup

Me.Resources.Add("RiaContext", RiaContext.Current)

Me.RootVisual = New MainPage()

End Sub

##### C#

private void Application\_Startup(object sender, StartupEventArgs e)

{

this.Resources.Add("RiaContext", RiaContext.Current);

this.RootVisual = new MainPage();

}

When the *RiaContext* is set as the *DataContext* of a control, the Authentication Service and User states becomes available for data binding.

<StackPanel Orientation="Horizontal"

DataContext="{StaticResource RiaContext}">

<TextBlock Text="User.Name" />

<TextBlock Text="{Binding User.Name}" />

<TextBlock Text="Authentication.IsBusy" />

<TextBlock Text="{Binding Authentication.IsBusy}" />

</Grid>

## Roles

### Server Tasks

#### Configure ASP.NET

Authentication Service roles are closely tied to the Roles framework provided by ASP.NET. The first step to enabling roles is to correctly configure the *web.config* file in the Web project. As above, the first few minutes of this ASP.NET [video](http://www.asp.net/learn/3.5-videos/video-45.aspx) provide a great introduction and the ASP.NET [wiki](http://wiki.asp.net/page.aspx/43/authentication-and-authorization/) provides guidance for this and other security-related tasks. To simply enable role support in ASP.NET, add the following to the *<system.web>* section of *web.config*.

<roleManager enabled="true" />

### Client Usage

After first completing the steps to enable authentication, and now completing the steps to enable roles, the Authentication Service will support role-based authorization. Roles are now available as part of the user state and can be accessed at *RiaContext.Current.User.Roles*. Also, using the *RiaContext*, roles can be bound to in XAML with *{Binding User.Roles}*.

## Profile

### Server Tasks

#### Configure ASP.NET

Authentication Service profile is closely tied to the Profile framework provided by ASP.NET. The first step to enabling profile is to correctly configure the *web.config* file in the Web project. There are some good resources at [MSDN](http://msdn.microsoft.com/en-us/library/2y3fs9xs.aspx) to help you setup profile properties. For parity with the next step, here’s a sample profile configuration.

<profile enabled="true">

<properties>

<add name="PhoneNumber"/>

</properties>

</profile>

#### Create a DomainService

To successfully include user profile in the user state, we need to revisit the second step from the authentication section above. Instead of just defining an empty *User* class, we’ll begin to add additional profile information.

##### Visual Basic

<EnableClientAccess()> \_

Public Class AuthenticationService

Inherits AuthenticationBase(Of User)

End Class

Public Class User

Inherits UserBase

Dim \_phoneNumber As String

Public Property PhoneNumber() As String

Get

Return Me.\_phoneNumber

End Get

Set(ByVal value As String)

Me.\_phoneNumber = value

End Set

End Property

End Class

##### C#

[EnableClientAccess]

public class AuthenticationService : AuthenticationBase<User> { }

public class User : UserBase

{

public string PhoneNumber { get; set; }

}

### Client Usage

After completing the steps for authentication and profile, we’re ready to use profile on the client. Thanks to the code generation provided by the Domain Services framework, the profile properties we added in step 2 are available on the client. When working in code, the profile values will be available as part of the user state and accessible off *RiaContext.Current.User*.

##### Visual Basic

RiaContext.Current.User.PhoneNumber = "unlisted"

##### C#

RiaContext.Current.User.PhoneNumber = "unlisted";

Again using *RiaContext* we can directly bind to profile properties in XAML.

<TextBox Text="{Binding User.PhoneNumber, Mode=TwoWay}" />

As above, *RiaContext.Current.Authentication.LoadUser* can be used to refresh the state of the user. Changes made to the user state can be persisted at any point with *RiaContext.Current.Authentication.SaveUser*.

# How to Restrict Access to DomainService Operations

Restricting access to domain operations is a critical requirement for many applications. The .NET RIA Services framework provides the ability to restrict access to domain services at the service and operation level via declarative attributes.

## Procedure

The following steps describe how to restrict access to domain operations.

### Prerequisites:

* You have an existing domain service that defines one or more operations.

### Steps

1. Mark the domain operations you want to restrict with one of the following attributes:
   1. **RequiresAuthenticationAttribute**

Requires that the current user is authenticated prior to beginning a domain service operation.

* 1. **RequiresRolesAttribute.**

Requires that the current user is authenticated and is a member of the roles specified prior to beginning a domain service operation.

Authorization attributes applied at the service level are applied to all domain operations within.

## Example

The example below demonstrated how domain operations can be annotated with authorization attributes.

|  |
| --- |
| namespace NorthwindExample  {  using System;  using System.Linq;  using System.Web.DomainServices;  using System.Web.DomainServices.LinqToEntities;  using System.Web.Ria;  [EnableClientAccess]  public class NorthwindService :  LinqToEntitiesDomainService<NorthwindEntities>  {  // Only authenticated users can retrieve  // Employee information.  [RequiresAuthentication]  public IQueryable<Employee> GetEmployees()  {  return this.Context.Employees;  }  // Only authenticated users in the "Manager"  // role can promote Employees.  [RequiresRoles("Manager")]  public void PromoteEmployee(Employee employee)  {  // Congratulations!  employee.Salary += 5000M;  employee.HasConvenientParkingSpace = true;  }  }  } |

The example below demonstrates how a domain service type can be annotated with authorization attributes. Here, all domain operations inherit the authorization requirements defined at the service level.

|  |
| --- |
| namespace NorthwindExample  {  using System;  using System.Linq;  using System.Web.DomainServices;  using System.Web.DomainServices.LinqToEntities;  using System.Web.Ria;  // Here, we apply authorization requirements at the service level.  [RequiresRoles("Administrator")]  [EnableClientAccess]  public class AnotherDomainService : DomainService  {  // ... all operations within require the "Administrator" role ...  }  } |

## Extensibility

In addition to the RequiresAuthenticationAttribute and RequiresRolesAttribute already provided, you can define your own authorization attributes by deriving from the AuthorizationAttribute class and overriding the Authorize(IPrincipal) method. You can then apply your custom attributes to domain operations.

|  |
| --- |
| namespace NorthwindExample  {  using System;  using System.Linq;  using System.Web.DomainServices;  using System.Web.DomainServices.LinqToEntities;  using System.Web.Ria;  [EnableClientAccess]  public class NorthwindService :  LinqToEntitiesDomainService<NorthwindEntities>  {  // Using custom attributes instead  [MyCustomAuthorizationAttribute]  public IQueryable<Employee> GetEmployees()  {  return this.Context.Employees;  }  }  // ...  public class MyCustomAuthorizationAttribute : AuthorizationAttribute  {  public override bool Authorize(IPrincipal principal)  {  // Perform custom authorization logic ...  return true;  }  }  } |

## See Also

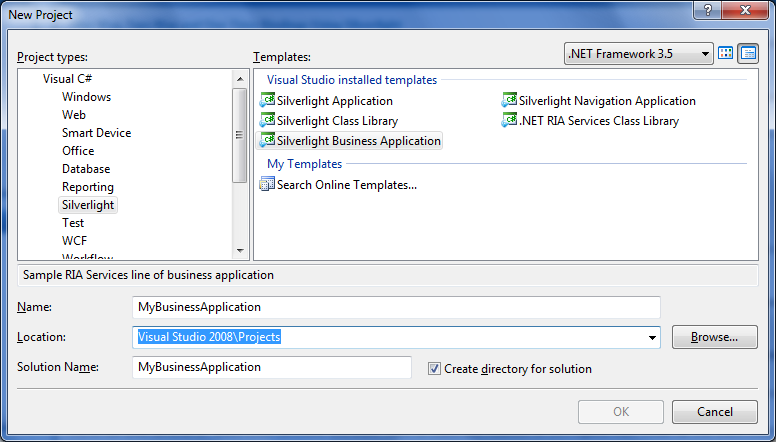
* Understanding N-Tier Silverlight Application Projects
* Authentication, Roles and Profiles

# Inside the Business Application Template

The Silverlight Business Application Template provides users with a starting point to build their Business Application on. The template builds on the Silverlight Navigation Application and using .NET RIA Services provides support for Authentication and User Registration.

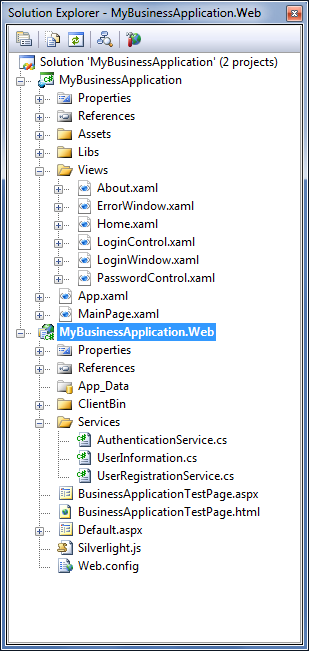
## Getting Started

The Silverlight Business Application Template shows up under SilverlIght Templates only on machiens that have .NET RIA Services installed.

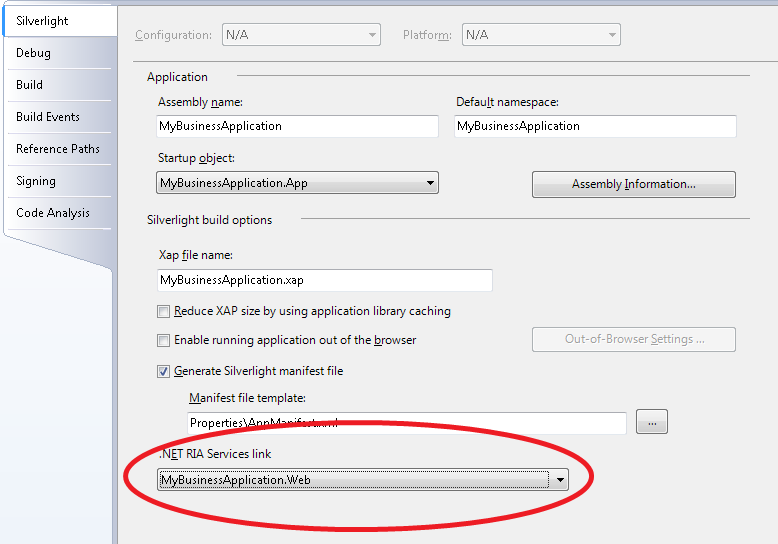


### Project Structure

Clicking ‘OK’ on the dialog above causes the following project layout to be created.



.NET RIA Services are enabled for the project above i.e. there is a CodeGen link between the Silverlight Business Application Client and the Server. This link is represented by the following drop down in the Silverlight Applications Property Page …



In addition to files that are part of the Silverlight Navigation Template, the following files are added by the Business Application template:

In the Server Project there is a **[Services]** Folder that contains .NET RIA Domain Services for Authentication and UserRegistration.

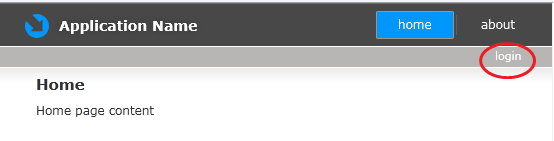
In the Silverlight project the **[Views]** folder has new Authentication and UserRegistration controls.

There is also a **[Libs]** folder in the Silverlight Project that contains the ActivityControl.dll and System.Windows.Controls.Data.DataForm.Toolkit.dll. Both of these assemblies are part of the Silverlight 3 Toolkit and have been included here for use by the Template.

### Controls

***Login Control***

On first running the application you will notice a Login link. Once the user has logged in it morphs into a message and Logout link. The source for this control can be found under [Views]\LoginControl.xaml

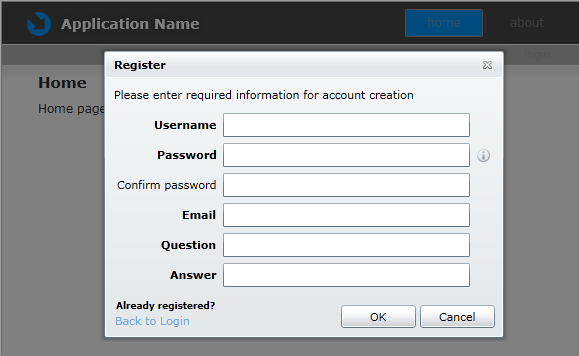


******

***Login Window***

Clicking on the Login link brings up the Login Window. Clicking on the ‘Register Now’ link morphs the Window into the User Registration ViewState. The source for this control can be found under [Views]\LoginWindow.xaml





## .NET RIA Services Usage

### Forms Authentication

Authentication support for the Business Application template is build using the .NET RIA Services Authentication Stack. The default Authentication mode for the Business Application is **Forms Authentication**, but the template supports **Windows Authentication** as well.

### Server

The Web App Project generated by the Business Application Template has Forms Authentication enabled in the Web.Config : <authentication mode="Forms"/>

The Authentication Domain Service shown below is added to the Web App to expose the Authentication functionality to the client.

**C#**

namespace MyBusinessApplication.Web

{

[EnableClientAccess]

public class AuthenticationService : AuthenticationBase<User>

{

}

public class User : UserBase

{

// public string MyProfileProperty { get; set; }

}

}

**VB**

<EnableClientAccess()> \_

Public Class AuthenticationDomainService1

Inherits AuthenticationBase(Of User)

End Class

Public Class User

Inherits UserBase

'Private myProfile As String

'Public Property MyProfileProperty() As String

' Get

' Return myProfile

' End Get

' Set(ByVal value As String)

' myProfile = value

' End Set

'End Property

End Class

The Authentication Domain Service class above is an empty class that derives from the ‘AuthenticationBase’ class which does all the heavy lifting. The AuthenticationBase class ships as part of .NET RIA Services and has built in support for the default ASP.NET Forms and Windows Authentication providers.

If the Application Developer wants to use their own Custom Authentication, they can do so by implementing IAuthentication<User> instead of deriving from Authentication Base.

The Business Application Template does not do define any UserSettings for the Application. However if the Application Developer wants to add User Settings to the Application, she can do so by first defining new Settings in the ASP.Net Profile Service and then by adding the same settings to the User class above.

### Client

The Silverlight Application is registered to consume the Authentication Domain Service due to the following lines of XAML added to the App.xaml by the template.

<Application.ApplicationLifetimeObjects>

<app:RiaContext>

<app:RiaContext.Authentication>

<appsvc:FormsAuthentication/>

</app:RiaContext.Authentication>

</app:RiaContext>

</Application.ApplicationLifetimeObjects>

Now the Application Developer can consume the Authentication Service on the client as follows: RiaContext.Current.Authentication.MethodName()

Similarly if there were UserSettings defined they can be obtained as follows: RiaContext.Current.User.MySetting

### Windows Authentication

To enable the Windows Authentication for the Application the Developer has to do the following:

1. Edit the Web.Config to enable Windows Auth: <authentication mode="Windows"/>
2. Modify the App.xaml as below

<Application.ApplicationLifetimeObjects>

<app:RiaContext>

<app:RiaContext.Authentication>

**<appsvc:WindowsAuthentication/>**

</app:RiaContext.Authentication>

</app:RiaContext>

</Application.ApplicationLifetimeObjects>

The Business Application will now support Windows Auth.

### User Registration Service

The “New User Registration” functionality is exposed to the Silverlight Client using a Domain Service. The service exists in the [Services]\ UserRegistrationService.cs file on the server. The UserRegistrationService internally uses the ASP.NET membership provider to implement the UserRegistration functionality.

# Writing a https Enabled DomainService

.NET RIA Services makes running an application entirely over HTTP or HTTPS simple. The scheme the Silverlight application (.XAP) is opened in will be the default scheme for that application. Sometimes it’s desirable to use SSL to communicate a subset of the data the application presents. Data that contains sensitive customer or business information should be protected in transport. To support this scenario, a DomainService can be configured to require communications happen over HTTPS. The service will deny all requests made over HTTP. Also, the generated DomainContext will make all requests using HTTPS even if the application was opened using HTTP.

## Brief Example

Let’s examine how to enable secure forms authentication. If you’re using forms authentication for an internet-facing application, it is strongly recommended that you use a secure connection.

We’ll start by creating a DomainService for authentication.

[EnableClientAccess]

public class Authentication : AuthenticationBase<User> { }

This generates a DomainContext on the client that calls into the data service that is available on the same scheme as the application. To change this default behavior, we’ll just update the client access attribute.

[EnableClientAccess(RequiresSsl = true)]

public class Authentication : AuthenticationBase<User> { }

Now the Authentication service can only be accessed using https. The application can be loaded on either http or https but the generated DomainContext will always attempt to reach the service over https.

## Further Information

Here are some pointers to help you enable SSL for your website. The above example creates a cross-domain scenario so you’ll have to set up a client access policy. Also, you’ll need IIS to host and test the secure endpoint. Finally you’ll need a valid (or trusted) SSL certificate. If you need a little help, here are some resources you may find useful.

* <http://timheuer.com/blog/archive/2008/10/14/calling-secure-services-with-silverlight-2-ssl-https.aspx>
* <http://timheuer.com/blog/archive/0001/01/01/silverlight-cannot-access-web-service.aspx>
* <http://weblogs.asp.net/scottgu/archive/2007/04/06/tip-trick-enabling-ssl-on-iis7-using-self-signed-certificates.aspx>
* <http://blogs.msdn.com/bags/archive/2009/01/23/self-signed-certificates-on-iis-7-the-easy-way-and-the-most-effective-way.aspx>

# Code Generated Hook Points

## Overview

Generated code provides hook points in the form of partial methods that are invoked at runtime by the framework code if the developer has implemented them. If the developer has not implemented them, these methods do not exist at runtime and are therefore not called by the framework.

The primary purpose of the hook points is to allow easy customization of generated code logic. Developers can add customization or initializing logic to the generated domain context or entity. This customization logic can define what their DomainContext/Entities do when a particular action is taken. An additional benefit is that the entities can now have computed properties.

## DomainContext Hookpoints

### OnCreated()

A partial method will be generated on the client for the developer to use to add custom logic to be executed when the DomainContext is instantiated.

**Method Signature:** partial void OnCreated()

**Called from:** DomainContext Constructors

## Entity Hookpoints

### OnCreated()

**Method Signature:** partial void OnCreated()

**Called from:** Entity Constructors

### OnLoaded(bool isIntialLoad)

**Method Signature:** protected virtual OnLoaded(bool IsIntialLoad)

**Called from:** Either of the following places:

1. When the entity is loaded from the server and deserialized for the first. The IsInitalLoad parameter will be true.
2. When the entity is merged IsInitalLoad will be false. Merging occurs after the entity is deserialized from the server but already exists on the client. The entity is merged with the server copy.

**Special notes:** The isInitialLoad should be passed to the base constructor.

protected override void OnLoaded(bool isInitialLoad)

{

// Custom logic here

base.OnLoaded(isInitialLoad);

}

### On[PropertyName]Changing([PropetyType] value)

The On[PropertyName]Changing partial method will be called in property setters. It will be called after validation but before the value is set.

**Method Signature:**  partial void On[PropertyName]Changing([propertyType] value)

### On[PropertyName]Changed()

The On[PropertyName]Changed partial method will be called in the property setters of each property. It will be called just after the value is set and before RaiseDataMemberChanged.

**Method Signature:** partial void On[PropertyName]Changed()

An entity with the property Id of type int would have the following hook points:

partial void OnIdChanging(int value);

partial void OnIdChanged();

### On[CustomMethodName]Invoking

A new partial method will be generated for the developer to use to add their custom logic and it will be called whenever the custom method is called but before it is invoked.

**Method Signature:**

partial void On[CustomOperation]Invoking([typeOfParameter2] [nameOfParameter2], [typeOfParameter3] [nameOfParameter3], ...)

### On[CustomMethodName]Invoked

A new partial method will be generated for the developer to use to add their custom logic and it will be called whenever the custom method is called after it has been invoked.

**Method Signature:** partial void On[CustomOperation]Invoked()

For example if an Employee entity had a CustomOperation named PromoteEmployee with the following signature:

public void PromoteEmployee(Employee entity, string newTitle, int newSalary)

Then the following hook points would be created in the DomainContext:

partial void OnPromoteEmployeeInvoking(string newTitle, int newSalary);

partial void OnPromoteEmployeeInvoked();

### Is[CustomMethodName]Invoked

**Type:** bool

Is[CustomMethodName] Invoked is a bool read-only property and will be available on the generated entity class. It will be marked False by default. The value will be true when the custom method is called and to false when the custom method is executed on SubmitChanges() or RejectChanges().

**Modifier:** public

**Signature**: public bool is[CustomMethodName] Invoked;

## Scenarios

### DomainContext Initialization

This example assumes you are using a DomainService named EmployeeService that has a Query method that uses an Employee entity.

The Employees are loaded when a new EmployeeContext is created. In order to load the Employees when a new EmployeeContext is created follow these steps:

1. Add a new Code File
2. Create a partial class called EmployeeContext
3. Declare a partial method for OnCreated that calls load using the GetEmployeesQuery.

public partial class EmployeeContext : DomainContext

{

partial void OnCreated()

{

this.Load(this.GetEmployeesQuery());

}

}

### Entity Initialization

One might want to keep track of the person that added an Employee. On creating a new Employee the CreatedBy will be populated with the user id. To do so follow these steps:

1. Add a new code file
2. Create a partial class called Employee
3. Declare a partial method for OnCreated and add logic to update the CreatedBy

public partial class Employee : Entity

{

partial void OnCreated()

{

this.CreatedBy = App.UserService.UserName;

}

}

Entities can also be initialized by a serializer (when being loaded from the server). In this case the virtual method OnLoaded should be used.

public partial class Employee : Entity

{

protected override void OnLoaded(bool isInitialLoad)

{

// Custom logic here

if (isInitialLoad)

{

this.LastAccessedBy = App.UserService.UserName;

}

base.OnLoaded(isInitialLoad);

}

### 

### Computed properties

The addition of code gen hook points now makes adding change notifications to computed properties a lot simpler. Please see the section “How to add computed properties”.

# How to add computed properties

This section explores how client side computed properties can be defined and displayed in the UI.

## Sample Setup

The example code in this document uses .NET RIA Services [walkthrough project](http://code.msdn.microsoft.com/RiaServices) as the base project and builds on top of it. To start working on this example, open the walkthrough project.

## Adding client side computed property

Let’s assume that we want to compute the ‘total off hours’ for each employee and display it in the UI. Total off hours is calculated as a sum of vacation hours and sick hours for an employee.

In CTP11, client side code generated file has hook points added to entities via partial methods that enable users to add custom code to customize the behavior based on the app scenario. We will use the OnPropertyXChanged partial method to raise property changed event to let the UI refresh the values whenever the dependent properties change.

Steps:

* Add a new code file to the Silverlight project.
* Add a new property TotalOffHours to employee entity, which has only a getter and returns the sum of SickLeaveHours and VacationHours.
* Add two partial methods OnSickLeaveHoursChanged and OnVacationHoursChanged to employee entity that raises propertychanged event on the computed property. Code snippet is given below.

namespace HRApp.Web

{

public partial class Employee

{

public int TotalOffHours

{

get { return this.SickLeaveHours + this.VacationHours; }

}

partial void OnSickLeaveHoursChanged()

{

this.RaisePropertyChanged("TotalOffHours");

}

partial void OnVacationHoursChanged()

{

this.RaisePropertyChanged("TotalOffHours");

}

}

}

## Data binding computed property to UI

Now we want to display this new computed property in the UI. To do this, open EmployeeList.xaml in design view and add the following XAML to Dataform’s EditTemplate section.

<dataForm:DataField Label="Total Off Hours">

<TextBox Text="{Binding TotalOffHours, Mode=OneWay}" />

</dataForm:DataField>

Note that this has binding marked as one-way as the user can’t modify this property and it’s computed.

Now if you run the project you will see ‘Total Off Hours’ text box appear for employee entity in the data form and whenever you edit the vacation hours or sick leave hours members, this textbox gets updated to reflect the newly computed value.

# Understanding N-tier Class Libraries

.NET RIA Services makes it easy to create an N-tier application within a single Visual Studio solution that includes an ASP.NET Web application project and one or more Silverlight client applications.

A special form of project-to-project reference known as the “RIA Link[[17]](#footnote-17)” informs Visual Studio these projects are related and can share business logic. Custom build tasks generate the necessary client proxy types in the Silverlight client project to share this business logic.

The July 09 Preview of .NET RIA Services contains a new feature known as “.NET RIA Services class libraries” that permits packaging business logic into N-tier class library components. This section describes what these libraries are and how you can use them.

## Overview

### Building N-tier applications using .NET RIA Services

Creating an N-tier application using .NET RIA Services is described in greater detail in the section “Understanding N-tier Silverlight Application Projects”. For the sake of this discussion, we can represent one of these applications graphically like this:

.NET RIA Services Application

Web App

Silverlight App

**RIA Link**

The Web App is an ASP.NET Web application project and is often referred to as the “mid-tier project.” The Silverlight App is a Silverlight application project and is often referred to as the “client-tier project.” Both projects live in the same Visual Studio solution.

The “RIA Link” arrow is the diagrammatic representation for the “Enable .NET RIA Services” property of the Silverlight client project and is described [below](#_What_does_the_1). This link enables sharing business logic between the tiers.

**RIA Link**

The solution structure shown above is the default when creating a new Silverlight application using File New Project and checking the “Enable .NET RIA Services” checkbox.

### Using class libraries on the mid-tier prior to the July 09 preview

The default project structure illustrated above is convenient when first getting started. All DomainService types and shared code added to the Web project become available to the Silverlight project automatically.

But it has the downside of encouraging developers to place all the business logic from potentially many domains into a single project. Developers who wanted to develop different sets of reusable domain logic generally packaged them into independent class libraries, like this:

.NET RIA Services App

Web App

Silverlight App

**RIA Link**

**.NET Class Library**

**.NET Class Library**

**.NET Class Library**

**.NET Class Library**

In other words, developers used existing Visual Studio templates to create class libraries and referred to them from the Web project. The.NET RIA Services infrastructure knew to treat DomainService types found in class libraries the same way it treated those in the Web project itself.

Prior to the July 09 Preview, this was the only supported way to use class libraries, but it had some drawbacks, including:

* Shared *source files* in the class libraries were not visible to the Silverlight project(s), making it difficult for the client to share source code from the class libraries.
* Proxy types were generated into the Silverlight project(s) for *all* DomainServices visible to the Web application. There was no way to restrict the Silverlight project(s) to use only a subset. This made it awkward for one Web application to support different Silverlight applications that each required different subsets of the available DomainServices.

### New in July 09 Preview – the .NET RIA Services Class Library

This July 09 preview of .NET RIA Services adds a new component to the developer’s toolbox; namely the ***.NET RIA Services Class Library***. In the same way that a .NET RIA Services *application* is an N-tier application packaged as a single unit, the .NET RIA Services class library is an *N-tier class library* packaged as a single unit.

This new component allows an application project structure like this:

**.NET RIA Services Class Library**

.NET RIA Services App

Web App

Silverlight App

**.NET Library**

**Silverlight Library**

**RIA Link**

In other words, the same RIA Link previously supported between *application* projects is now also supported between *class library* projects. It is the same link (“Enable .NET RIA Services”), and it has all the same semantics of business logic sharing, client proxy class generation, etc.

Effectively then, the new .NET RIA Services class library concept can be thought of as a single N-tier class library component. It has a mid-tier piece and a client-tier piece, but it is conceptually a single component.

There is no restriction on the number of the class libraries an application can use or on the number of applications that can reuse a class library.

The previous model continues to work, meaning the developer can still continue to use the RIA Link between the application projects. The new feature of the July 09 Preview is simply an additional tool in the developer’s toolbox. It provides some useful new capabilities, including:

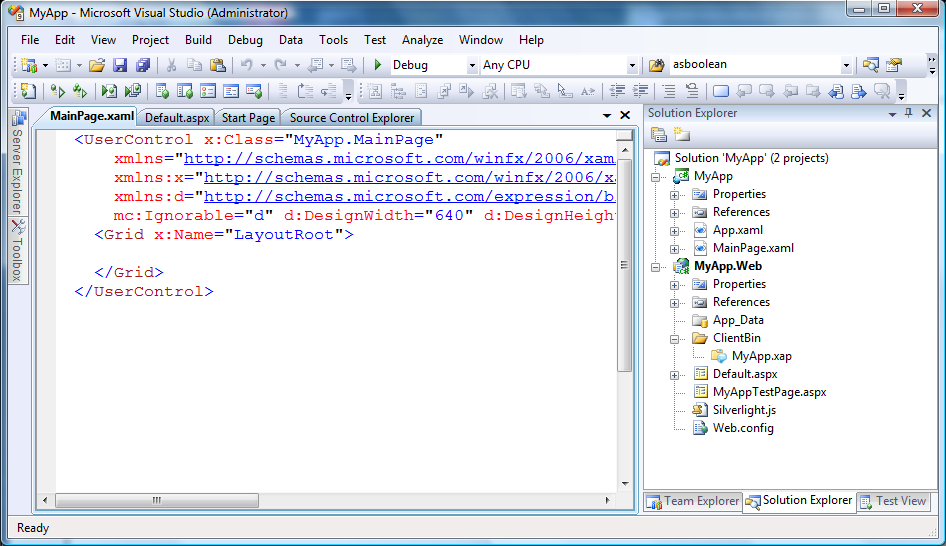
* The server and client portions of a single domain of data can be developed and packaged as a single component and reused in multiple applications.
* Client proxy code generation and source code sharing now occurs in only one place – between the halves of this component, and not into every Silverlight application.
* Multiple Silverlight applications within a single Web application project can refer to just the class libraries they need. No longer does every Silverlight client need to see all the business logic exposed by the mid-tier.

## Using the .NET RIA Services Class Library

This section describes how to create a new .NET RIA Services class library and use it in your application. We’ll create a new one, hook it into our existing application, and add a new DomainService to see how the plumbing works.

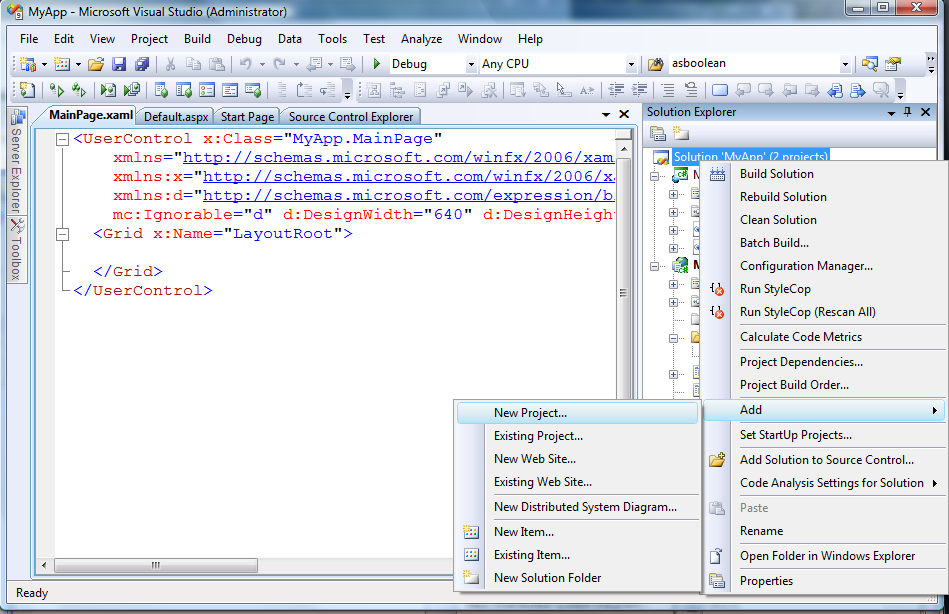
### Start with an existing Web application

Let’s assume you start with an existing Web application project (created via File | New Project | Silverlight Application[[18]](#footnote-18)) that looks like this:

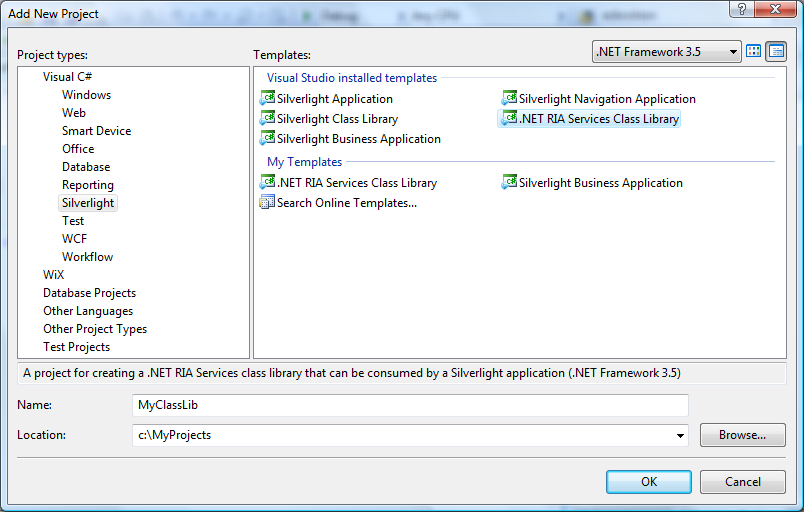


### Create a new .NET RIA Services class library using the new template

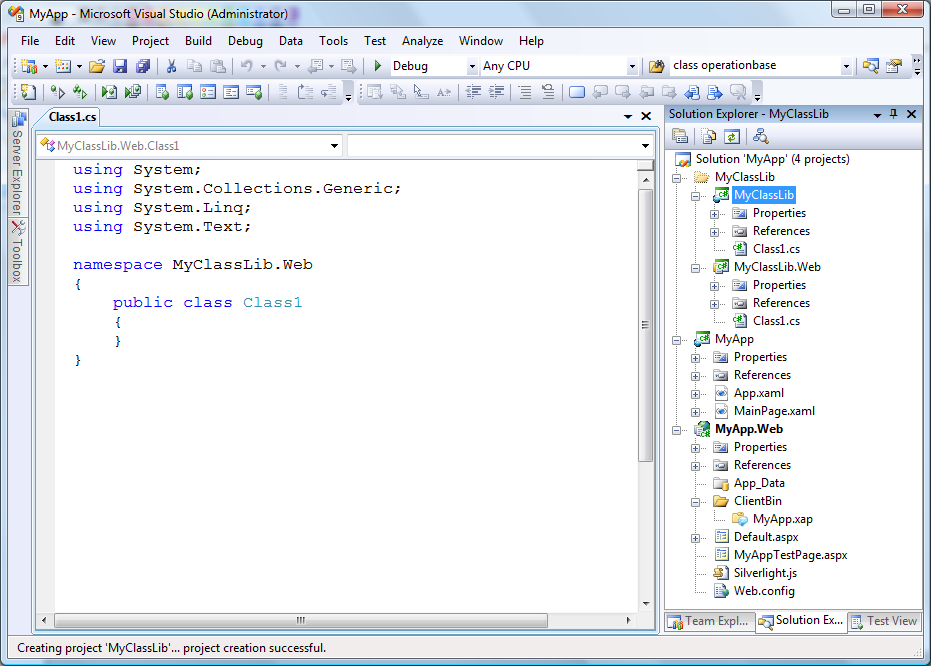
The July 09 preview provides a new project template to create N-tier class libraries. To create a new .NET RIA Services class library, right-click the solution node and ask to Add | New Project… like this:



When the Add New Project wizard appears, you will find a new .NET RIA Services Class Library project template under the Silverlight category. Select that and give the library a name, like this:



When you click OK, the project template will update the solution to look like this:



The template added these items to the solution:

1. A new solution folder (MyClassLib) to contain both class library projects[[19]](#footnote-19)
2. The mid-tier class library project (MyClassLib/MyClassLib.Web)
3. The client-tier class library project (MyClassLib/MyClassLib)

These are just normal class libraries that you could have constructed yourself with the existing class library templates *but with these important defaults:*

1. The “Enable .NET RIA Services” option in the Silverlight library project has been enabled to create a RIA Link with the mid-tier class library
2. A build dependency has been inserted into the solution to force the mid-tier library to build before its associated client-tier library.

It is not necessary to place the .NET RIA Services class library into the same solution as the application. It was done here for simplicity only. In practice, organizations may choose to keep these in separate solutions. After all, they are just class library projects and can be packaged however you like.

### Add references to the class libraries

To use these class libraries in your application, you must manually add assembly (or project) references from the application projects to the new library projects. In this simple example, we would do the following 2 steps:

1. Right-click the MyApp.Web project and use “Add Reference…” to add a reference to the MyClassLib.Web library project
2. Right-click the MyApp (Silverlight) project and use “Add Reference…” to add a reference to the MyClassLib (Silverlight) library project.

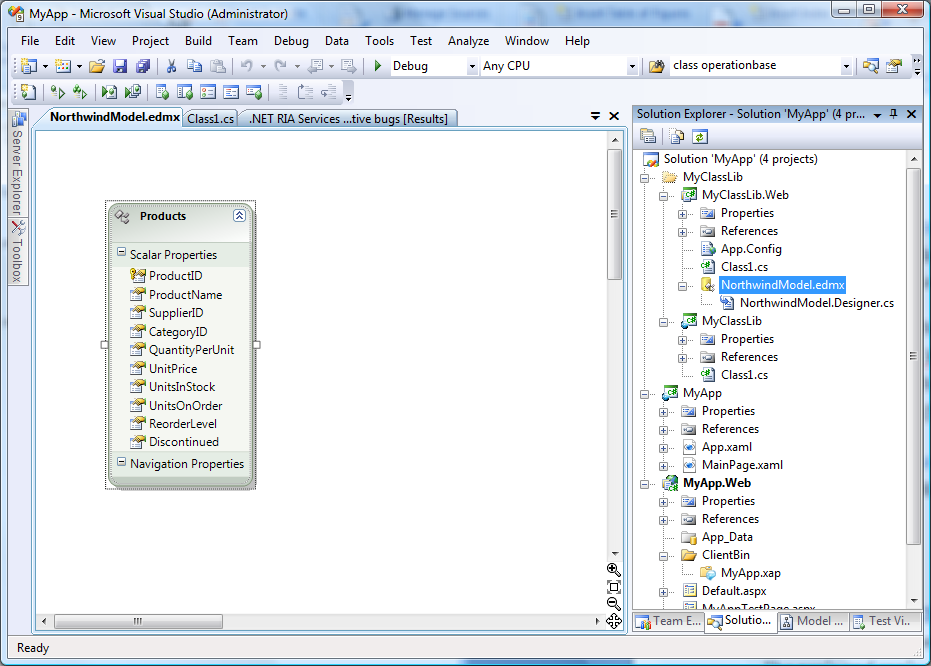
If you fail to add a reference to the Silverlight library the Silverlight application will simply not have access to those entities or business logic, which in some cases may be precisely what you want. In other words, it is your choice which Silverlight libraries the Silverlight application will use.

But if you fail to add a reference from the Web project to the mid-tier class library, incoming HTTP requests to the mid-tier to access the respective DomainService will not know how to find it, and the client application will receive a “Resource not found” exception.

The general guideline is to add a reference from the Web application project to all mid-tier class libraries it needs to expose to *all* its client applications, regardless whether the Web application project accesses those DomainServices directly. But for each Silverlight client project, add references to only those client-tier class libraries you need.

### Add a model to the mid-tier library

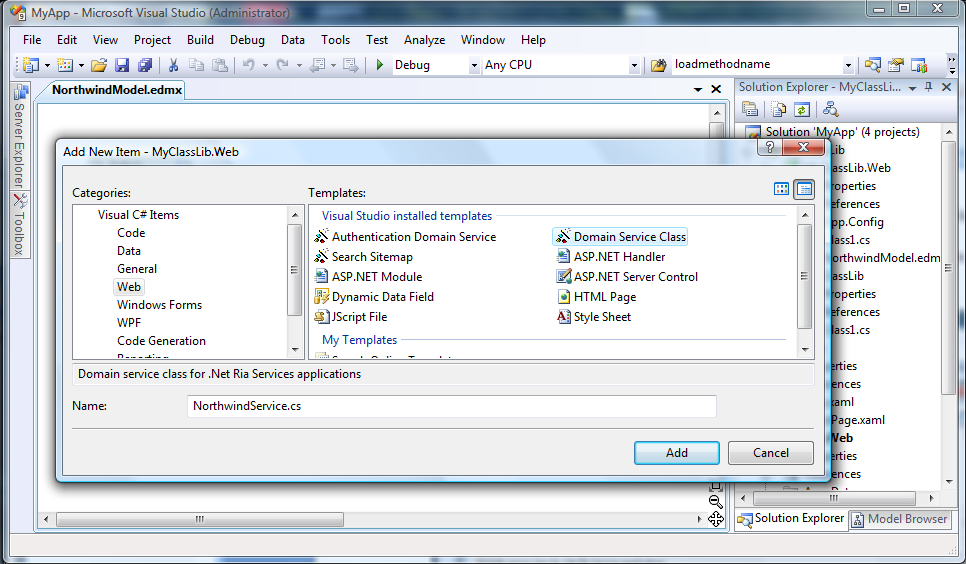
At this point, the creation of the .NET RIA Services class library is complete. It is fully wired up and ready to use. But for the sake of completeness, let’s add a model to the mid-tier library and see how code generation occurs. In the following diagram, we’ve added an Entity Framework model to expose the Products table from the Northwind database:



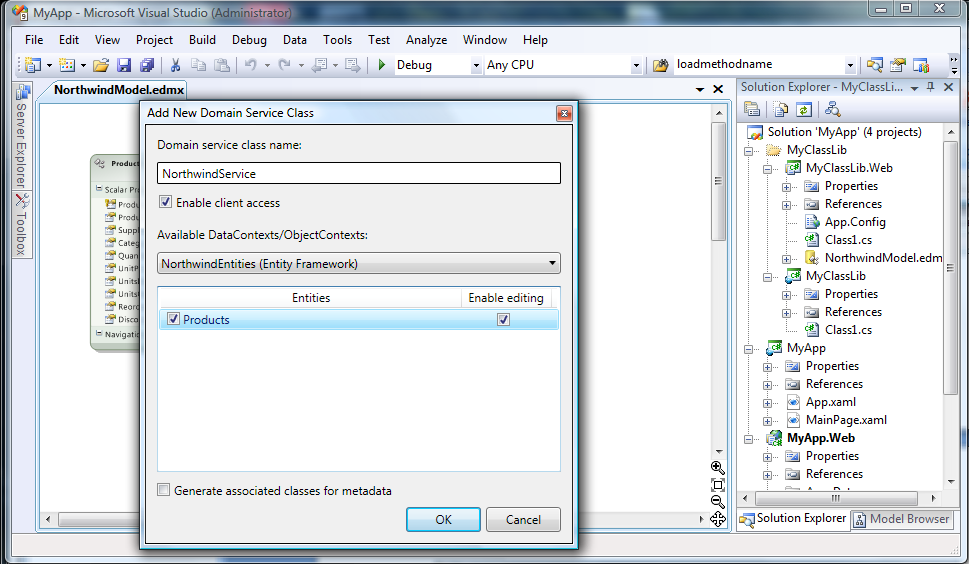
### Create a DomainService in the mid-tier library

And now let’s create a new DomainService in the mid-tier library to use this model.

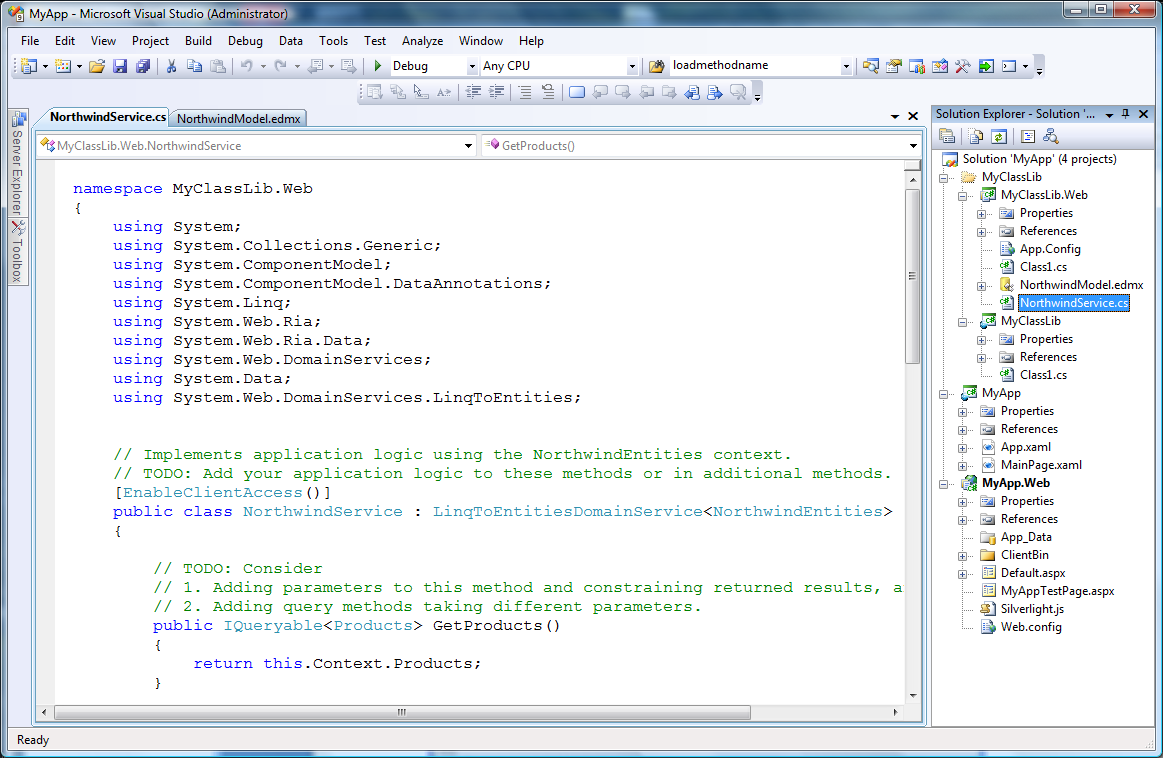
Build the project and use Add New Item to add a new Domain Service class to the MyClassLib.Web project, like this:



Using the wizard associated with this template, we’ll choose to expose the Products entity and to allow it to be edited:

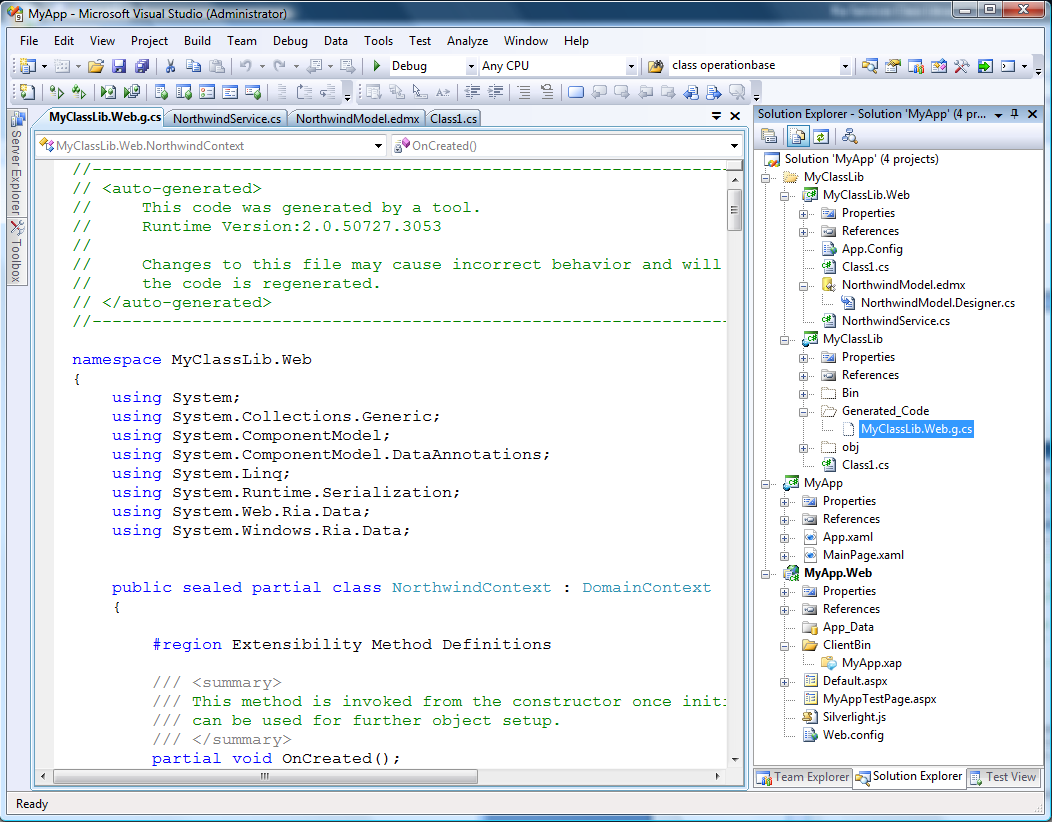


This will produce a new DomainService class in the mid-tier library:



### Build the solution

When you build the solution, you will now find that .NET RIA Services has generated client proxy types into the client-tier library[[20]](#footnote-20).



Equally important is the fact that these client proxy classes have *not* been generated in the Silverlight application project. They exist only in the client-tier library. However, the Silverlight client application can still use them as it would any other type in a class library.

### Use the generated code from the client application

Because the Silverlight client application has an assembly reference to the client-tier half of the .NET RIA Services class library, it can now start using the code that was generated into that library[[21]](#footnote-21).

In this example, the developer could add the following code to their Silverlight application:

using System.Windows.Ria;

using System.Windows.Ria.Data;

using MyClassLib.Web;

…

NorthwindContext context = new NorthwindContext();

context.Load(context.GetProductsQuery());

…

## Possible project structures

Given these new capabilities, it is possible to organize projects in several different ways. This section describes some of the possible project structures you might consider, along with some of the benefits or trade-offs each entails.

### Simplest application – no class libraries

This is the simplest project structure and the one you will get by default unless you explicitly create class libraries. It shows 2 domain services and 2 client applications to illustrate the implications of using this kind of structure.

.NET RIA Services Application

Silverlight App1

Web App

**RIA Link**

**Shared code1**

**DomainService1**

**Shared code2**

**DomainService2**

**DomainContext1**

**DomainContext2**

**Shared code1**

**Shared code2**

Silverlight App2

**DomainContext1**

**DomainContext2**

**Shared code1**

**Shared code2**

Key:

**Generated code**

**Developer written code**

**RIA Link**

To translate this diagram:

* The Web application project contains 2 domain services and 2 sets of shared code
* This solution has 2 Silverlight applications, each linked to the Web project with a RIA Link[[22]](#footnote-22)
* When the solution is built, .NET RIA Services:
  + Generates proxy types for *both* domain services into *both* Silverlight projects
  + Copies *both* sets of shared code files into *both* Silverlight projects

Advantages:

* Everything is in one solution
* Shared source code is visible to the client projects

Disadvantages:

* The DomainServices and shared code are not reusable by other solutions
* *All* Silverlight applications with a RIA Link to the Web project receive proxy classes and shared code for *all* its DomainServices – they cannot select only a subset.
* All Silverlight applications generate their own separate copies of the proxy classes and shared code

This is the simplest .NET RIA Services solution structure, and it is suitable when first getting started or for applications that involve a small number of DomainServices or Silverlight applications. But as the application grows, you may want to consider reorganizing it using one of the structures below.

### Mid-tier class libraries

Using the following alternate project structure, the developer has moved the domain services and shared code into separate class libraries to permit reuse in other applications. These are just conventional class libraries used by the mid-tier, not the new .NET RIA Services class libraries. This structure was achievable using earlier versions of .NET RIA Services and involved the use of existing class library project templates.

.NET RIA Services Application

.NET Class Lib2

Silverlight App1

Web App

**RIA Link**

**Shared code2**

**DomainService2**

.NET Class Lib1

**Shared code1**

**DomainService1**

**Shared code3**

**DomainContext1**

**DomainContext2**

**Shared code 3**

Silverlight App2

**DomainContext1**

**DomainContext2**

**Shared code 3**

**RIA Link**

This new structure changes the organization of the solution, making the domain service class libraries reusable by other projects, but it really does not alter code generation. All Silverlight applications with a RIA Link to the Web project continue to receive proxy classes for all the domain services, regardless where they live. There is one notable difference – the Silverlight client projects no longer receive copies of the shared source code because it is not visible to them.

Advantages:

* The domain service class libraries can be reused in other applications

Disadvantages:

* The shared code (specifically \*.shared.cs or \*.shared.vb) in the class libraries is not propagated to the Silverlight client during code generation[[23]](#footnote-23).
* Source code intended to be shared by the client needs to be moved into the Web project, losing some of the benefits of packaging the domain services separately.
* All Silverlight clients continue to see all domain services and continue to generate duplicate copies of the proxy classes

This solution structure offers some benefits in terms of reuse on the mid-tier, but it does not alter the structure or behavior of the Silverlight client projects. Frankly, the shortcomings of this structure were the motivation behind the development of .NET RIA Services class libraries and the solution structure shown below.

### Using the new .NET RIA Services class libraries

The introduction of the .NET RIA Services class libraries allows a more flexible project structure, like this:

**.NET RIA Services Class Lib**

.NET RIA Services App

.NET Class Lib2

Silverlight App1

Web App

**RIA Link**

**Shared code2**

**DomainService2**

**.NET RIA Services Class Lib**

.NET Class Lib1

**RIA Link**

**Shared code1**

**DomainService1**

SL Class Lib1

Silverlight App2

**Shared code 1**

**DomainContext1**

SL Class Lib2

**Shared code 2**

**DomainContext2**

In this structure, the developer has created a separate .NET RIA Services class library for each domain service and its associated shared code. Code generation and sharing occurs strictly between the halves of each .NET RIA Services class library. The application projects have added references to their respective halves of the libraries.

Note in particular that Silverlight App1 has references to both Silverlight class libraries because it needs to access both sets of business logic, but Silverlight App2 has a reference only to one of them because that’s all it needs. In other words, this project organization allows the developer to choose which domain services each Silverlight application will use. For example, you can imagine a customer-facing Silverlight application and an internal-facing Silverlight application sharing a common Web application but exposing different business logic by using different DomainServices.

Advantages:

* The domain services class libraries can be reused in other applications
* Silverlight applications can use only the subset of business logic they need
* No duplicate code generation occurs
* Shared source files flow naturally into their respective Silverlight library

Disadvantages:

* Slightly more complex project structure that requires the developer to be aware of a new concept; the .NET RIA Services class library

### Effects of mixing the RIA Link between the Web project and class libraries

The astute reader may have noticed one tiny omission in the project structure above – *the RIA Link was missing between the Silverlight apps and the Web app*. Why was that? And what happens if we put it back?

The short answers are:

***Why was the RIA Link removed?*** It was no longer necessary. The class libraries contain the generated proxy types and shared code, and the application can use them via assembly references.

***What happens if we put it back?*** This link is a request to see *all* business logic visible to the Web project, so the Silverlight client project will once again pick up proxy classes for *all* DomainServices visible to the Web project.

Let’s add the RIA Link back to Silverlight App2 and observe what happens. We show this to illustrate how the mechanisms work so you can make an informed choice. To make this change, we’ll select Project Properties for Silverlight App 2, and set the “.NET RIA Services link” to point to the Web application (in other words, we create a “RIA Link” between Silverlight App2 and the Web app):

**.NET RIA Services Class Lib**

.NET RIA Services App

.NET Class Lib2

Silverlight App1

Web App

**RIA Link**

**Shared code2**

**DomainService2**

**.NET RIA Services Class Lib**

.NET Class Lib1

**RIA Link**

**Shared code1**

**DomainService1**

SL Class Lib1

Silverlight App2

**Shared code 1**

**DomainContext1**

SL Class Lib2

**Shared code 2**

**DomainContext2**

**RIA Link**

**Shared code 2**

**DomainContext2**

Building the solution now generates code into Silverlight App2. Why?

Remember that the RIA Link is an all-or-nothing option. It effectively says “my client project wants to share *all* business logic visible to the project I point to.” By pointing to the Web application, Silverlight App2 now becomes aware of both domain services. The code generator detected that DomainService1 was already visible through the SL Class Lib1 and did not need to be generated a second time. However, because Silverlight App2 had no reference to allow it see generated code for DomainService2, the code generator created a new copy for it.

To illustrate this further, we’ll make one more small change to the project structure; we’ll add an assembly reference from Silverlight App2 to the SL Class Lib2. Again, this is to better illustrate the mechanisms so you can choose what you want:

**.NET RIA Services Class Lib**

.NET RIA Services App

.NET Class Lib2

Silverlight App1

Web App

**RIA Link**

**Shared code2**

**DomainService2**

**.NET RIA Services Class Lib**

.NET Class Lib1

**RIA Link**

**Shared code1**

**DomainService1**

SL Class Lib1

Silverlight App2

**Shared code 1**

**DomainContext1**

SL Class Lib2

**Shared code 2**

**DomainContext2**

**RIA Link**

After we build the solution, those redundant copies of the generated code are no longer in Silverlight App2. The code generator discovered Silverlight App2 had access to this generated code through its new assembly reference, so the redundant copies could be removed.

We mention this somewhat subtle topic to drive home 2 important concepts:

1. The RIA Link is “all or nothing”. Don’t create a RIA Link between 2 projects unless you want the client project to see ***all*** domain services visible to its linked project.
2. When the code generator detects existing paths to generated code for a particular domain service, it will not create redundant copies.

### View model class libraries

The RIA Link is not restricted to only app🡪app or library🡪library configurations. It can be applied directly between a Silverlight class library and the Web Application project, like this:

.NET RIA Services App

Silverlight App1

Web App

**RIA Link**

Silverlight Class Lib

**Shared code**

**DomainContext**

**Shared code**

**DomainService**

Silverlight App2

This can be described as the *view model configuration*. The Silverlight class library into which the client proxy classes are generated can expose a view model to one or more Silverlight application projects. This view model configuration would allow additional client-side business or presentation logic to be encapsulated into its own class library, and the application(s) would not need to know these implementation details to use view models the library exposes.

### Recommended project structures

We offer the following recommendations to keep matters simple:

* If you are building relatively simple applications, stick with using a RIA Link between the Silverlight client application and the Web application. Don’t bother using the new .NET RIA Services class libraries.
* However, if your application involves multiple domain services or Silverlight applications, and you are thinking about reuse, consider using .NET RIA class libraries and *remove the RIA Link* between the client and Web *application* projects.

## Miscellaneous

### What does the “Enable .NET RIA Services” checkbox do?

When .NET RIA Services is installed, creating a new Silverlight project[[24]](#footnote-24) shows a checkbox called “Enable .NET RIA Services.”

We refer to this as the “RIA Link” and show it graphically in this document like this:

**RIA Link**

When you select this option, a property is inserted into the Silverlight project file:

<LinkedServerProject>..\MyClassLib.Web\MyClassLib.Web.csproj</LinkedServerProject>

This link is a special form of project-to-project references and always points from a Silverlight client project (application or class library) to some other .NET project (application or class library).

The rules for this link are:

* It can appear only in Silverlight client projects
* There can be only one per Silverlight client project
* It cannot point to other Silverlight projects
* Multiple Silverlight client projects can link to the same .NET project
* A Silverlight *application* project cannot link directly to a .NET class library project

The semantics of how this link is handled at compile time are:

* If there are files named \*.shared.cs or \*.shared.vb in the project pointed to by this link, copy them verbatim into the client project Generated\_Code folder and add them to the in-memory list of files to compile
* Examine all the assemblies known to the project pointed to by this link, and for every DomainService type marked with the EnableClientAccessAttribute, generate into the client project a corresponding DomainContext type and proxy classes for all the exposed entity types.

The details of how the RIA Link drives code generation are described in “”.

### Special issues when placing DAL models in class libraries

Once you start using class libraries to isolate the Domain Services, it is only another small step to place the DAL (Data Access Layer) models into their own separate class libraries too. But this organization encounters some issues you need to understand. These issues are not unique to .NET RIA Services.

When you place an Entity Framework or LINQ-to-SQL data model directly into a Web application project, the web.config will be modified automatically to contain the appropriate connection string. When the application is launched, this connection string is used to open the respective database connection.

However, if you place one of these models into a class library, the respective connection string is placed in an app.config file in that library. If a Web application project uses these class libraries, it will be unaware of their app.configs and therefore the connection strings will not be available.

Whenever you place DAL models into class libraries, you need to manually copy the connection string portion of the app.config to the Web application project’s web.config.

A sample of the relevant section is highlighted below:

<?xml version="1.0"?>

<configuration>

<connectionStrings>

<add name="NorthwindEntities" connectionString="metadata=res://\*/Model1.csdl|res://\*/Model1.ssdl|res://\*/Model1.msl;provider=System.Data.SqlClient;provider connection string=&quot;Data Source=.\SQLEXPRESS;AttachDbFilename=|DataDirectory|\NORTHWND.MDF;Integrated Security=True;Connect Timeout=30;User Instance=True;MultipleActiveResultSets=True&quot;" providerName="System.Data.EntityClient"/>

</connectionStrings>

</configuration

If you fail to copy this connection string into the web.config, the DAL model will fail to open its respective database, and your application will encounter a runtime exception.

### Special issues when creating DomainServices in class libraries

If you use the DomainService class wizard (Add New Item … Web | Domain Service class) to create a new DomainService class, it will automatically update the project’s config file to register an HTTP handler for incoming .NET RIA Services requests. Only one handler registration is necessary for all domain service types.

But if you use this wizard to create a DomainService class inside a class library, it can update only the app.config. Just as you did with the connection string section above, you need to manually copy that section of the app.config to the Web project’s web.config.

The necessary sections are highlighted below. Notice that there are 2 sections that need to be copied.

<?xml version="1.0"?>

<configuration>

<configSections>

<system.web>

<httpHandlers>

<add path="DataService.axd" verb="GET,POST" type="System.Web.Ria.DataServiceFactory, System.Web.Ria, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35" validate="false"/>

</httpHandlers>

</system.web>

<system.webServer>

<handlers>

<add name="DataService" verb="GET,POST" path="DataService.axd" type="System.Web.Ria.DataServiceFactory, System.Web.Ria, Version=2.0.0.0, Culture=neutral, PublicKeyToken=31BF3856AD364E35"/>

</handlers>

</system.webServer>

</configuration>

Note: the second section is necessary when deploying the app using IIS7.

Failure to copy these sections into the web.config will manifest a “Resource not found” exception at runtime when the Silverlight client application attempts to access its respective DomainService.

## Summary

The July 09 Preview of .NET RIA Services supports a new feature; .NET RIA Services Class Libraries. Using this feature, different DomainServices and their associated shared business logic can be packaged into N-tier class libraries which can be reused in multiple applications.

This feature allows the developer to organize N-tier applications into reusable components and allows the mid-tier and client-tier applications to use only the components they need.

# How to Work with Multiple DomainContexts

Applications occasionally want to display data from a mixture of data sources. .NET RIA Services enables a set of such scenarios by supporting cross-DomainContext references that allow entities to express associations with entities originating from different DomainContext types.

One example where this might be useful is an e-commerce website that uses a proprietary ordering system but with products originating from a third-party DomainService. In this scenario, we have data originating from two distinct sources but want to create an association between the two.

## Procedure

### Prerequisites:

* You already have 2 (or more) DomainServices in your web project.
* You have 1 (or more) entities that you would like to relate to external entities exposed by different DomainService.
  + You have full control over the entity declaration or the entities are declared as partial types.
* All of the entities that you intend to share across DomainContext boundaries will exist in the client application.

### Steps:

#### Server Project

1. Create a new *partial* class to extend an entity in your server project.
2. Add a new property to the entity type declaration. This property type should be of the entity type that you would like to expose. (That is, the property type should be of an entity type that is exposed by a different DomainService.)
3. Mark the new property with the following attributes:
   1. ExternalAttribute (System.Web.Ria.Data)

*This indicates to the framework that we do not need to generate the entity type in the client code as another DomainService is already doing so.*

* 1. AssociationAttribute (System.ComponentModel.DataAnnotations)

*Provide the appropriate association name and keys.*

1. Build your project to refresh the client generated code.

#### Client Project

1. Using an instance of the DomainContext that exposes the entity extended in step #1 above; invoke the AddReference generic method passing in an entity type reference and a DomainContext instance that exposes the same entity type. (Alternatively, a non-generic AddReference overload exists allowing for the entity type to be specified as a method argument.)

Now that the required DomainContext associations are setup, you can load and access cross-DomainContext entities.

**Note**: Entities that are referenced across DomainContext instances are not loaded on demand. External entity references will be null until all appropriate external entities have been loaded in their respective origin DomainContexts. (I.e., be sure to load the relevant entities in each linked DomainContext instance before accessing cross-DomainContext entity associations.)

**Note:** cross-DomainContext associations are unidirectional. To create a bidirectional association, you will need to update entities on each side of a cross-DomainContext association using the steps described above.

## Basic Example

In this example, we have two existing domain service types: **MyOrdersService** and **ProductCatalogDomainService**. These domain services return **Order** and **Product** types respectively. Here we will create an external relationship that allows **Order** to reference **Product**.

### Server Side

1. First, in the server project we extend the type declaration of **Order** by adding an additional property called “Product” that is of type **Product**.

**Note**: If the Product property was generated from frameworks such as ADO.NET Entity Framework, you can still extend the type declaration of **Order** by adding the new property in a partial type declaration.

1. We then annotate this property with the **ExternalAttribute** (System.Web.Ria.Data). This is used to indicate that this entity type exists externally.
2. We also need to annotate this property with an **AssociationAttribute** (System.ComponentModel.DataAnnotations) to describe the external entity association.

|  |
| --- |
| namespace MyOrders  {  using System;  using System.ComponentModel.DataAnnotations;  using System.Web.Ria.Data;  public partial class Order  {  // ...  // Other interesting Order properties exist  // in another source file.  // ...  [External]  [Association(  "Order\_Product", // Unique ssociation Name  "ProductID", // The key property on this side (Order.ProductID)  "ID", // The key property on the other side (Product.ID)  IsForeignKey = true)] // Indicates that this end is the foreign key  public Product Product { get; set; }  }  } |

1. Build the solution to update the generated code in the client project.

### Client Project

1. The generated client-side entity representation of **Order** should now have a new **Product** property. (It should also be annotated with an **ExternalAttribute** and **AssociationAttribute**.)
2. Within the client project code, we’ll create an instance of each DomainContext type and setup an association between the two by registering an external reference for the **Product** type.

|  |
| --- |
| public void InitializeDomainContexts()  {  // Instantiate our providers  MyOrdersContext myOrders = new MyOrdersContext();  ProductCatalogContext productCatalog = new ProductCatalogContext();  // Add 'productCatalog' as a reference for external lookups.  // The 'myOrders' instance will now be able to access the  // Products contained in the 'productCatalog' instance.  myOrders.AddReference<Product>(productCatalog);  // Start loading data in each DomainContext...  // so that the cross-DomainContext associations will be populated.  } |

Some things to be aware of:

* An external reference cannot be added to a DomainContext for a Type that is already exposed by that DomainContext. (E.g., you cannot add a ‘Product’ reference to the ‘ProductCatalogContext’ since it is already exposed by the DomainContext.)
* Entities that are references externally are *not* loaded on demand.
* Properties that reference external entities will return null until said external entities are loaded in one of the referenced DomainContext types.

## Advanced Example

The example above shows how to make use of the existing code generation infrastructure to define cross-DomainService/DomainContext relationships on the server that are propagated down to the client. One drawback to this approach is that it creates introduces additional entity properties that may not have any meaning in the server object model.

It is also possible to create cross-DomainContext references and keep these relationships isolated in your client project. This example is intended for advanced users.

Here, we’ll proceed with the two existing domain service types described above (**MyOrdersService** and **ProductCatalogDomainService**). These domain services return **Order** and **Product** types respectively. Here we will create an external relationship that allows **Order** to reference **Product** by modifying the client side code.

### Client Project

1. Add a new class declaration called “Order.cs” and extend the existing partial definition of the Order entity that was automatically generated. (Be sure to use the appropriate matching namespace.)
2. Inside of this partial definition for Order, add a new public property of type Product and named “Product”.
3. Add a new property to this partial Order declaration, following the pattern shown below.

|  |
| --- |
| public partial class Order  {  // (1) EntityRef<Product> field used by property.  private EntityRef<Product> \_product;  // (2) Product property must be marked with  // [External] and [Association(...)].  [External]  [Association("Order\_Product", "ProductID", "ID", IsForeignKey = true)]  public Product Product  {  // Getter only  get  {  // Lazily loaded.  if ((this.\_product == null))  {  // Instantiate a new EntityRef<Product> using  // the filter predicate defined below.  this.\_product = new EntityRef<Product>(  this,  "Product",  this.FilterProduct);  }  return this.\_product.Entity;  }  }  // (3) Filter predicate used above to return the appropriate  // external Product instances.  private bool FilterProduct(Product entity)  {  return (entity.ProductID == this.ProductID);  }  } |

1. After adding the property definition as shown above, step #6 in the “Basic Example” can be followed.
   1. **See Also**

* Understanding N-Tier Silverlight Application Projects
* Understanding Silverlight Client Code Generation

# How to unit test business logic

One of the common tasks that [.NET RIA Services](http://code.msdn.microsoft.com/RiaServices) developers have to undertake is testing their mid-tier business logic code. Mid-tier code typically uses a data access layer (DAL) like LINQ to SQL or LINQ to Entities for persisting data. However directly coupling the business logic code to the DAL will pose challenges for unit testing and causes tests to depend on the database. One possible solution to avoid this problem is to use the repository pattern and write business logic code to go against repository.

The following steps demonstrate an implementation of a domain service class using a repository to make it unit testing friendly. The example code uses the .NET RIA services [walkthrough application](http://code.msdn.microsoft.com/RiaServices/Release/ProjectReleases.aspx?ReleaseId=2387) as its starting point and is modified to use LINQ To SQL as DAL.

## Sample Setup

To start, follow the walkthrough steps to create a sandbox application that we can use to explore the APIs:

* Create the new Silverlight application, naming it “TestSample”, configuring all the other project settings as in the walkthrough
  + Make sure the SL app has a reference to the System.Windows.Controls.Data assembly
* Add a copy of the AdventureWorks DB referenced in the walkthrough to the App\_Data folder of the Web project
* Add a new LINQ to SQL data model to the Web project by selecting “LINQ to SQL classes” in the “Add New” dialog
  + Name the DBML file “AdventureWorks.dbml”
  + Use Server Explorer to add a new Data Connection to the AdventureWorks\_Data.mdf file
* Use LINQ to SQL designer to add the Product entity, by dragging the Employee table from Server Explorer onto the designer surface
* Build the solution
* Add a new DomainService class to the web app by following the steps under “Creating a DomainService” in the .NET RIA Services overview document
  + Name the DomainService “OrganizationDomainService”
  + Select the Product entity and make sure the “Enable Editing” checkbox is checked
  + Select the “Generate associated classes for metadata” checkbox
* Build the solution
* Finally bind the UI and load some data by adding the following XAML and code behind (new code highlighted):

<UserControl x:Class="TestSample.MainPage"

xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"

xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"

xmlns:data="clr-namespace:System.Windows.Controls;assembly=System.Windows.Controls.Data"

Width="Auto" Height="Auto">

<Grid x:Name="LayoutRoot">

<data:DataGrid Height="Auto" Width="Auto" x:Name="employeesGrid"></data:DataGrid>

</Grid>

</UserControl>

**STEP 1:** Create an interface that defines the necessary methods for performing basic CRUD operations on the repository

namespace HRApp.Web.Repository

{

public interface IRepository<T>

{

IQueryable<T> Query();

void Add(T entity);

void Remove(T entity);

void Attach(T current, T original);

void Attach(T current);

void SubmitChanges();

}

}

**STEP 2:** Provide concrete implementation of IRepository for LINQ to SQL. This is done by performing the repository operations on LINQ to SQL’s DataContext as shown below.

namespace HRApp.Web.Repository {

public class LinqToSqlRepository<T> : IRepository<T> where T : class

{

private DataContext context;

public LinqToSqlRepository(DataContext ctxt)

{

this.context = ctxt;

}

#region IRepository<T> Members

public IQueryable<T> Query()

{

return this.context.GetTable<T>();

}

public void Add(T entity)

{

this.context.GetTable<T>().InsertOnSubmit(entity);

}

public void Remove(T entity)

{

this.context.GetTable<T>().DeleteOnSubmit(entity);

}

public void Attach(T current, T original)

{

this.context.GetTable<T>().Attach(current, original);

}

public void Attach(T current)

{

this.context.GetTable<T>().Attach(current);

}

public void SubmitChanges()

{

this.context.SubmitChanges();

}

#endregion

}

}

**STEP 3:** Implement the DomainService class and use the repository instead directly calling into DAL layer in domain operation implementation. The code below implements an Organization DomainService which uses an employee repository

[LinqToSqlMetadataProvider(typeof(AdventureWorksDataContext))]

[EnableClientAccess()]

public class OrganizationDomainService : DomainService

{

private IRepository<Employee> employeeRepository;

public OrganizationDomainService(IRepository<Employee> empRepository)

{

this.employeeRepository = empRepository;

}

public IQueryable<Employee> GetEmployee()

{

return this.employeeRepository.Query().OrderBy(e => e.EmployeeID);

}

public IQueryable<Employee> GetSalariedEmployee()

{

return this.employeeRepository.Query().Where(e => e.SalariedFlag == true).OrderBy(e => e.EmployeeID);

}

public void InsertEmployee(Employee employee)

{

// Modify the employee data to meet the database constraints.

employee.HireDate = DateTime.Now;

employee.ModifiedDate = DateTime.Now;

employee.VacationHours = 0;

employee.SickLeaveHours = 0;

employee.rowguid = Guid.NewGuid();

employee.ContactID = 1001;

employee.BirthDate = new DateTime(1967, 3, 18);

this.employeeRepository.Add(employee);

}

public void UpdateEmployee(Employee currentEmployee)

{

Employee originalEmployee = this.ChangeSet.GetOriginal(currentEmployee);

this.employeeRepository.Attach(currentEmployee, originalEmployee);

}

public void DeleteEmployee(Employee employee)

{

this.employeeRepository.Attach(employee);

this.employeeRepository.Remove(employee);

}

public void ApproveSabbatical(Employee current)

{

this.employeeRepository.Attach(current);

current.CurrentFlag = false;

}

#region Base class Overrides

protected override void PersistChangeSet(ChangeSet changeSet)

{

this.employeeRepository.SubmitChanges();

}

#region

}

As you can notice from above code snippet, there are few things that are different from the original walkthrough sample:

1. OrganizationDomainService doesn't derive from the LinqToEntitiesDomainService, instead it derives from the base DomainService class

2. OrganizationDomainService has been applied with LinqToSqlMetadataProviderAttribute. This adds LINQ To SQL’s type descriptor to our domain service class and enables it to reference the LINQ To SQL generated entities (Employee entity in this case).

3. Query/Update methods now call into Repository methods instead of directly calling DAL’s context methods.

4. PersistChangeSet method has been overridden to call the Repository’s SubmitChanges method.

**STEP 4:** Write a factory class to create instance of the DomainService class and the repository and register it (in Global.asax.cs). This ensures that our factory method is used for domain service class creation by the framework.

public class OrganizationDomainServiceFactory : IDomainServiceFactory

{

public DomainService CreateDomainService(Type domainServiceType,

DomainServiceContext context)

{

AdventureWorksDataContext ctxt = new AdventureWorksDataContext();

IRepository<Employee> empRepository = new LinqToSqlRepository<Employee>(ctxt);

DomainService ds = (DomainService)Activator.CreateInstance(domainServiceType,

new object[] { empRepository });

ds.Initialize(context);

return ds;

}

public void ReleaseDomainService(DomainService domainService)

{

}

}

Here is the code to registier the factory:

public class Global : System.Web.HttpApplication

{

protected void Application\_Start(object sender, EventArgs e)

{

DomainService.Factory = new OrganizationDomainServiceFactory();

}

..

}

Now the app is ready for execution (there is no change required to the client side code of the walkthrough project).

**STEP 5:** Creating unit tests

For unit testing the domain service methods, all we need to do is write a mock repository that creates an in-memory instance of employees and pass it to DomainService.

Define a MockRepository base class (in your test project):

namespace HRApp.Web.Test

{

public abstract class MockRepository<T> : IRepository<T> where T : class

{

private Dictionary<Type, List<T>> inMemoryTables = new Dictionary<Type, List<T>>();

public List<T> Table

{

get

{

return inMemoryTables[typeof(T)];

}

}

protected void AddTable()

{

List<T> table = new List<T>();

inMemoryTables.Add(typeof(T), table);

}

#region IRepository<T> Members

public IQueryable<T> Query()

{

return this.Table.AsQueryable();

}

public void Add(T entity)

{

this.Table.Add(entity);

}

public void Remove(T entity)

{

this.Table.Remove(entity);

}

public void Attach(T current, T original)

{

}

public void Attach(T current)

{

}

public void SubmitChanges()

{

}

#endregion

}

}

Create a mock employee repository:

namespace HRApp.Web.Test

{

public class MockEmployeeRepository : MockRepository<Employee>

{

public MockEmployeeRepository()

{

Init();

}

void Init()

{

// populate in-memory data

this.AddTable();

Employee p1 = new Employee() { EmployeeID = 1, SalariedFlag = true };

Employee p2 = new Employee() { EmployeeID = 2, SalariedFlag = true };

Employee p3 = new Employee() { EmployeeID = 3, SalariedFlag = false };

this.Table.Add(p1);

this.Table.Add(p2);

this.Table.Add(p3);

}

}

}

The unit test code would look something like this:

[TestClass]

public class OrganizationDomainServiceTest

{

[TestMethod]

public void TestGetSalariedEmployee()

{

MockEmployeeRepository mockEmployeeRepository = new MockEmployeeRepository();

OrganizationDomainService orgDomainSvc = new OrganizationDomainService(mockEmployeeRepository);

IQueryable<Employee> salariedEmployees = orgDomainSvc.GetSalariedEmployee();

Assert.IsTrue(salariedEmployees.Count() == 2, "Salaried employee count does not match.");

}

[TestMethod]

public void TestInsertEmployee()

{

MockEmployeeRepository mockEmployeeRepository = new MockEmployeeRepository();

OrganizationDomainService orgDomainSvc = new OrganizationDomainService(mockEmployeeRepository);

Employee newEmployee = new Employee() { EmployeeID = 9, SalariedFlag = true };

orgDomainSvc.InsertEmployee(newEmployee);

Assert.AreEqual(DateTime.Today, newEmployee.HireDate.Date, "Hire date not as expected.");

}

}

}

Completed sources can be downloaded from: <http://code.msdn.microsoft.com/RiaServices>

# How to display localized error messages

This section explores how error messages can be localized by storing them in resource files and sharing them across tiers.

## Sample Setup

The example code in this document uses .NET RIA Services [walkthrough project](http://code.msdn.microsoft.com/RiaServices) as the base project and builds on top of it. To start working on this example, open the walkthrough project.

## Creating error messages as resource file

Let’s say we want to add a validation error as a resource for LoginID field.

* Create a new ‘Resources’ folder in the HRApp.Web project (server project)
* Add a new resource file to this folder and name it **ValidationErrorResources.resx**
* Double click on the .RESX file to bring up resource designer page
* Add a new string resource with **Name= LoginIDValidationError** and **Value= "LoginID field is required"**
* Change the access modifier to ‘Public’ by clicking on the ‘Access Modifier’ drop down UI and selecting ‘Public’ and save the project. This generates a **ValidationErrorResources** class in the HRApp.Web.Resources namespace.
* Open ‘OrganizationService.metadata.cs’ file and add the following ‘Required’ field validation to LoginID member. Specify the error message resource name and resource type values by setting the corresponding attribute members as shown below.

[Required(ErrorMessageResourceName = "LoginIDValidationError", ErrorMessageResourceType = typeof(ValidationErrorResources))]

public string LoginID;

Now we want to share this resource file in the Silverlight project (client project). To do this,

* Create a folder Web\Resources in the HRApp project (folder structure must match the resource file namespace on the server side)
* Select Resources folder and bring up Add Existing file dialog, browse to the server side resource file folder location
* Select ValidationErrorResources.resx and ValidationErrorResources.designer.cs files, and add them as link files to the Silverlight project. Save the project file
* Open HRApp.csproj file in notepad , locate the section where .designer.cs file is included and add the highlighted 3 lines to this section

<Compile Include="..\HRApp.Web\Resources\ValidationErrorResources.Designer.cs">

<AutoGen>True</AutoGen>

<DesignTime>True</DesignTime>

<DependentUpon>ValidationErrorResources.resx</DependentUpon>

<Link>Web\Resources\ValidationErrorResources.Designer.cs</Link> </Compile>

* Save the project file and reload the project in Visual Studio
* Build the solution and run

Now whenever the validation fails for the LoginID field the error message from the resource file is shown to the user. The resource file can now be customized to store locale specific error messages.

# Using the ASP.NET DomainDataSource

This walkthrough shows how to create a Web application that uses a domain data service.

The connection to the domain data service is obtained through a DomainDataSource control. This control shelters you from knowing the syntax that you need to interface with a specific data source. Moreover, it enables you to work against some business logic instead than directly against the Data Context or the Data Abstraction Layer (DAL).

## Prerequisites

In order to run the procedures in this topic, you will need:

* Microsoft Visual Studio 2008 Service Pack 1.
* A database, such as the SQL Server Northwind.mdf sample database.
* .NET RIA (Rich Internet Application) Services.

## Procedures

|  |
| --- |
| To create a Web application |
| 1. Start Visual Studio or Visual Web Developer. 2. In the File menu, click New Project.   The New Project dialog box is displayed.   1. In the Project Types list, select the programming language that you prefer to work in. 2. Under Visual Studio installed templates, select ASP.NET Web Application. 3. In the Name box, enter the name of the project.   For example, enter the name DomainDataSource.   1. In the Location box, enter the name of the folder where you want to keep the Web application.   For example, enter the folder name C:\WebApplications.   1. Check Create directory for solution. 2. Click OK. 3. Visual Studio creates the folder and the structure for the Web application. |

|  |
| --- |
| To add the database file to the Web application |
| 1. In Solution Explorer, right-click the App\_Data folder, and then click Add Existing Item.   The Add Existing Item dialog box is displayed.   1. Enter the location where you installed the NorthWind.mdf database file. 2. Then select the file. |

|  |
| --- |
| To create the data model using LINQ to SQL |
| 1. In Solution Explorer, right-click the Web application name and then click Add New Item. 2. Under Visual Studio installed templates, click LINQ to SQL Classes. 3. In the Name box, enter a name for the database model.   For example, enter the name Northwind.dbml.   1. Click Add.   The Object Relational Designer is displayed.   1. In the O/R Designer, click the Server Explorer link (Database Explorer in vwprvw). 2. In Server Explorer (Database Explorer), under Data Connections, expand the database file node and then the Tables node. 3. Drag the Categories, Products, and Suppliers tables into the O/R Designer. 4. Save and close the Northwind.dbml model file. 5. Compile the application. |

|  |
| --- |
| To create the domain data service |
| 1. In Solution Explorer, right-click the Web application name and then click Add New Item. 2. Under Visual Studio installed templates, click Domain Service Class. 3. In the Name box, enter a name for the service model.   For example, enter the name NorthwindService.cs.   1. Click Add.   The New Domain Service Class dialog box is displayed.   1. Select all the tables. They will be accessed with the domain data service. 2. Check the Generate associated classes for metadata. 3. The NorthwindService.cs file will contain the methods for the selected tables. 4. Save and close the file. |

|  |
| --- |
| To configure the domain data service |
| 1. In Solution Explorer, right-click the Reference folder and add the reference to the following assemblies:   System.ComponentModel.DataAnnotations  System.Web.DomainServices.WebControls  These assemblies can be found in the RIA Service folder at this location: \Program Files\Microsoft SDKs\RIA Services\v1.0\Libraries\Server.   1. Open the web.config file and add the following configuration setting:   <pages>  <controls>  <add tagPrefix="asp"  namespace="System.Web.DomainServices.WebControls"  assembly="System.Web.DomainServices.WebControls"/>  <controls>  </pages>   1. Save and close the web.config file. |

|  |
| --- |
| To test the domain data source |
| 1. In Solution Explorer, right-click the Web application name and then click Add New Item. 2. Under Visual Studio installed templates, click Web Form. 3. In the Name dialog box enter the page name.   For example, enter domaindatasource.aspx.   1. Add to the page the markup for the DomainDataSource control.   The following example shows the markup that defines a DomainDataSource control which connects to the NorthwindService and executes the GetProducts select method.  <asp:DomainDataSource ID="DataSourceID" runat="server"  DomainServiceTypeName="DomainDataSource.NorthwindService"  SelectMethod="GetProducts">  </asp:DomainDataSource>   1. Add a T:System.Web.UI.WebControls.GridView data-bound control and bind it to the DomainDataSource control defined in the previous step.   The following example shows the markup for a T:System.Web.UI.WebControls.GridView control bound to the DomainDataSource control.  <asp:GridView ID="GridView1" runat="server"  DataSourceID="DataSourceID">   1. Save the page. 2. In Solution Explorer right-click on the page and then click View in Browser.   You should see the Products table displayed in the page. |

# ADO.NET Data Services Integration

ADO.NET Data Services is a framework for building and consuming data-oriented web services that first shipped in the Microsoft .NET Framework 3.5 SP1. ADO.NET Data Services provides libraries, Visual Studio tooling, and a set of well-defined patterns that make it easy for services to expose CRUD operations on a data model as simple HTTP operations.

ADO.NET Data Services defines a protocol for interacting with data services based on HTTP and the popular AtomPub and JSON standards. Beyond these standards, the ADO.NET Data Services protocol defines formats for data payloads and an addressing scheme for resources. Because of its open and interoperable design, a wide variety of clients and servers are beginning to adopt the ADO.NET Data Services protocol. Together, these form a growing ecosystem of producers and consumers of data.

In many ways, the ADO.NET Data Services protocol resembles the communication between the client and service tiers of a Microsoft .NET RIA Services application. Our goal is to align the two technologies such that Microsoft .NET RIA Services uses the same protocol as ADO.NET Data Services. This will provide benefits to both technologies: Microsoft .NET RIA Services applications will interoperate with rest of the Data Services ecosystem (leveraging existing clients and tooling), and services built with ADO.NET Data Services will be able to leverage the business logic patterns, rich UI controls, and end-to-end developer experience provided by Microsoft .NET RIA Services.

We are still in the early stages of this alignment effort and this CTP contains an implementation of our progress to date. We envision this CTP being used in two main scenarios:

1. Extending an existing Microsoft .NET RIA Services application by adding an ADO.NET Data Service endpoint to enable interaction via the ADO.NET Data Services protocol
2. Extending an existing ADO.NET Data Service using the Microsoft .NET RIA Services patterns to add business logic.

This document provides a walk-through of the first of these two scenarios: how to add an ADO.NET Data Service to an existing Microsoft .NET RIA Services application. The second scenario is not covered by this document.

As you work through this document and use the CTP, please note that we are still early into the alignment process, so this work has not gone through as much design iteration or received as much attention from the QA team as have other parts of this CTP which are “go-live” ready. We do not recommend you use the ADO.NET Data Services integration in a production application. We are interested in your feedback on this CTP and the direction of the alignment.

## The Sample Application

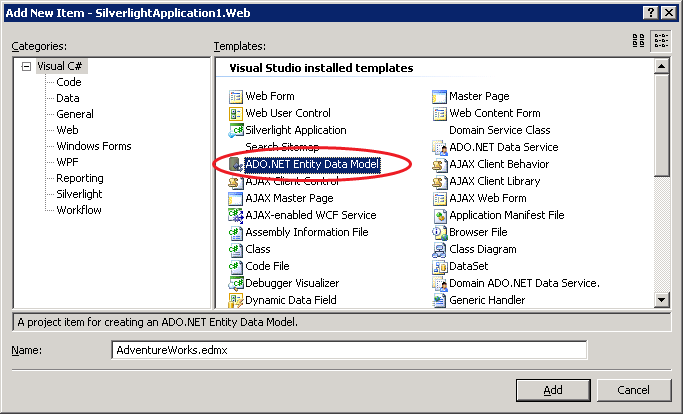
We’ll start with a small application as the basis for this walk-through. This application will just display data from the AdventureWorks sample database’s Products table in a grid and allow editing from the client.

To create the basic application, complete the following steps:

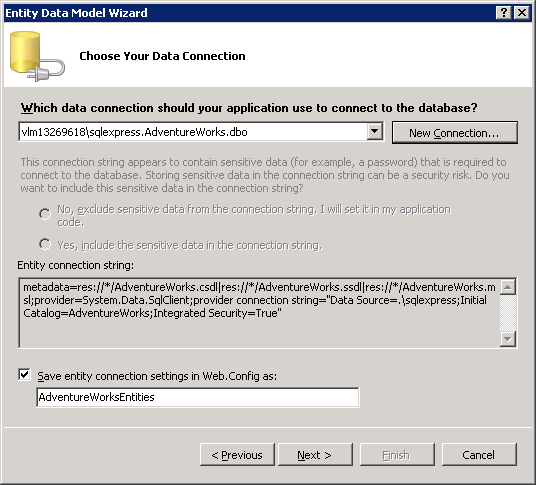
1. In Visual Studio 2008, create a new Silverlight Application project and call it ProductsWithDataService.
2. In the New Silverlight Application dialog, make sure the checkboxes for “Host the Silverlight application in a new Web site” and “Enable .NET RIA Services” are checked, then click OK.

This gives us a basic solution with a Silverlight application and a linked ASP.NET Web application. Next we’ll add an Entity Data Model to represent our data source:

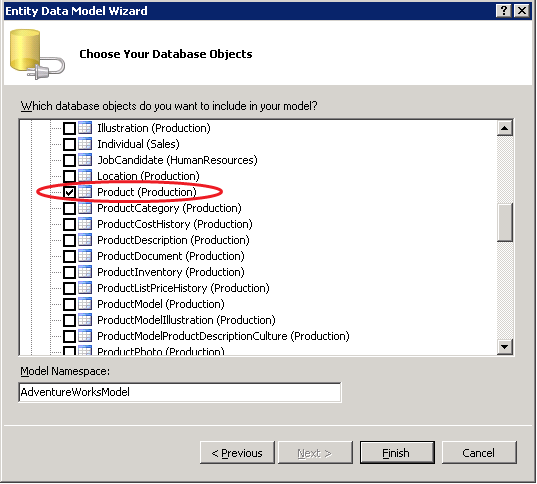
1. Right-click on the ProductsWithDataService.Web project in Solution Explorer, choose Add, then New Item.
2. From the Add New Item dialog, select ADO.NET Entity Data Model and change the model name to AdventureWorks.edmx. Click Add to start the Entity Data Model Wizard.



1. In the Entity Data Model Wizard, select Generate from database and then click Next.
2. Select (or create a new) connection to the AdventureWorks database on your SQL Server and make sure the text box beneath the “Save entity connection settings in Web.Config as:” checkbox is set to “AdventureWorksEntities”. Click Next.



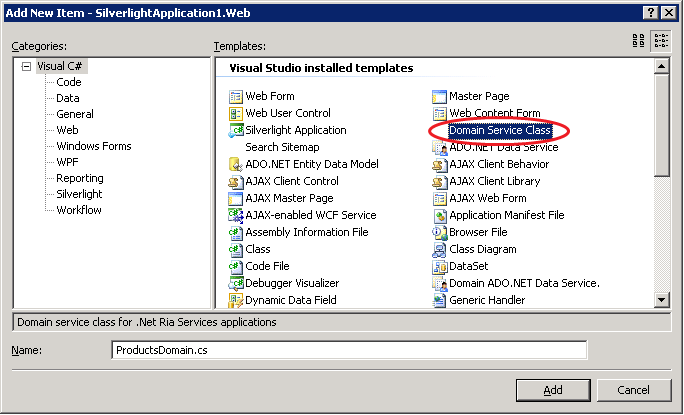
1. In the next page of the Entity Data Model Wizard, select only the Product table and make sure the Model Namespace is set to AdventureWorksModel. Click Finish to complete the Entity Data Model Wizard.



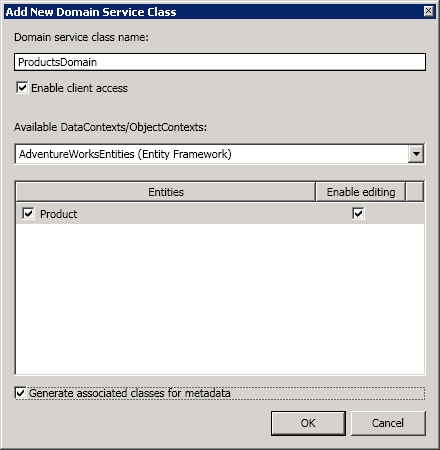
1. Build the project to compile the generated data classes into the project assembly.

Now that we have the data model set up, we can add a Domain Service class to encapsulate the business logic of the application:

1. Right-click on the ProductsWithDataService.Web project in Solution Explorer, choose Add, then New Item.
2. In the Add New Item dialog, select Domain Service Class, name it ProductsDomain.cs, then click Add.



1. In the Add New Domain Service Class dialog, make sure the “Enable client access” checkbox is checked and select the AdventureWorksEntities ObjectContext in the dropdown list. If you don’t see any items in this dropdown list, you probably forgot to build the project after adding the Entity Data Model – build the solution and try this step again.
2. Select the Product entity type and check the corresponding “Enable editing” box. Check the “Generated associated classes for metadata” box at the bottom of the dialog. With these options selected, the dialog should look like the figure shown below. Click OK to complete the Add New Domain Service Wizard.



The ProductsDomain class that the wizard generated has stubs for each of the CRUD operations against the Product entity set. For the purposes of this application, we’ll just use the default implementations of these stubs rather than customize them (in a full application, we could add custom logic to define the query, insert, update, and delete operations for the Product data). Because we checked the “Enable client access” box in the Add New Domain Service Class wizard, the ProductsDomain class has the [EnableClientAccess] attribute which exposes its methods to the client tier.

Now that we have a basic server working, we’ll focus next on the Silverlight UI. In this simple application, we’ll just use a grid to display (and allow edits to) the product data.

1. Open the MainPage.xaml file in the ProductsWithDataService Silverlight project.
2. Add the following XAML within the existing <Grid> element:

<Grid.RowDefinitions>

<RowDefinition Height="1\*" />

<RowDefinition Height="20" />

</Grid.RowDefinitions>

1. Drag a DataGrid from the Toolbox onto the XAML, placing it just beneath the row definitions you just added.
2. Add the attributes shown below (x:Name, AutoGenerateColumns, and Grid.Row) to the <DataGrid> element.

<data:DataGrid x:Name="productGrid" AutoGenerateColumns="true" Grid.Row="0" />

1. Open the MainPage.xaml.cs file.
2. Add the following using declarations at the top of the file:

using ProductsWithDataService.Web;

using System.Windows.Ria.Data;

1. Add code to the MainPage class so that it looks like the snippet below (new code shown in bold). This code creates an instance of our generated DomainContext class (ProductsDomain), calls the Load() method to load all the products data, and binds the grid to the loaded products entities.

public partial class MainPage : UserControl

{

**ProductsDomain \_domain = new ProductsDomain();**

public MainPage()

{

InitializeComponent();

this.\_domain.Load(this.\_domain.GetProductQuery());

this.productGrid.ItemsSource = \_domain.Products;

}

}

1. Build and run the application. You should see a simple grid displaying all the products data.

To complete the basic application, we’ll add a button that will save any changes made in the grid back to the data source.

1. Go back to MainPage.xaml and add the following XAML snippet right below the data grid:

<StackPanel Grid.Row="1" Orientation="Horizontal">

<Button Content="Save changes" Click="SaveChanges\_Click" />

</StackPanel>

1. Right-click anywhere in the declaration of the Click attribute and select Navigate to Event Handler. This generates the stub for the SaveChanges\_Click event handler in MainPage.xaml.cs.
2. Fill out the code in the SaveChanges\_Click handler as shown below:

private void SaveChanges\_Click(object sender, RoutedEventArgs e)

{

SubmitOperation submitOp = this.\_domain.SubmitChanges();

submitOp.Completed += delegate

{

if (submitOp.Error == null)

{

MessageBox.Show("Changes were saved successfully.");

}

else

{

MessageBox.Show(string.Format("Error while saving: {0}", submitOp.Error.ToString()));

}

};

}

1. Build and run the application. Make an edit somewhere in the grid and click the Save Changes button to see the changes persisted to the server.

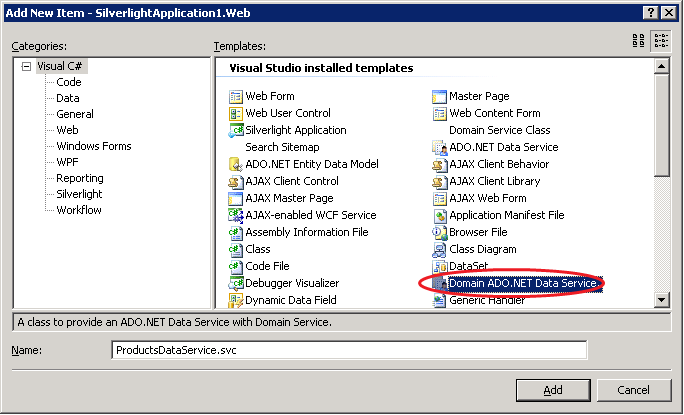
We now have a basic but functional application. At this point, we could enrich the application using the mechanisms provided by Microsoft .NET RIA Services: we could add validation attributes to the Product entity type, add custom validation code, or provide a richer user interface using other controls such as the DataPager and DataForm. We’re not going to do any of those things here (coverage of those topics is provided in other chapters). Instead, we’re going to focus on adding an ADO.NET Data Services interface to our application.

## Adding an ADO.NET Data Service

Up to this point, the client and server tiers of our sample application communicate using existing mechanisms in Microsoft .NET RIA Services. In this section, we will add an ADO.NET Data Service and observe the ADO.NET Data Services protocol on the wire. In the following section, we’ll modify the client tier to use the ADO.NET Data Services protocol.

To add the ADO.NET Data Service to the application, perform the following steps:

1. Right-click on the ProductsWithDataService.Web project in Solution Explorer, choose Add, then New Item.
2. In the Add New Item dialog, select Domain ADO.NET Data Service, name it ProductsDataService.svc, then click Add. (Note that there may be other Item Templates that have ADO.NET Data Service in their names – be sure to pick the one named Domain ADO.NET Data Service).



This will add the skeleton of an ADO.NET Data Service to the project and will open up the ProductsDataService.cs file containing the code that backs the service.

We need to make a few changes to the generated service code to get the ADO.NET Data Service running. First, we need to associate the service with a data source. In this case, we’ll use our Domain Service class, ProductsDomain, as the data source. ProductsDomain provides CRUD methods for the Product entity set; by configuring the ADO.NET Data Service to work over ProductsDomain, we will expose those CRUD methods as HTTP operations following the ADO.NET Data Services protocol.

After configuring the data source, we’ll have to specify access control rules for the service – for this simple example, we’ll just open up the service for full read/write access. In a real application, you would probably want to control access a little more carefully.

Complete the following steps to associate the ADO.NET Data Service with the ProductsDomain class and configure the Product entity set for read/write access:

1. Open the ProductsDataService.svc.cs file and notice that the service class, ProductsDataService, inherits from DataService<T>. In the generated code, the generic T parameter is missing and there is a TODO comment in its place:

public class ProductsDataService : DataService< /\* TODO: put your data source class name here \*/ >, IServiceProvider

1. Replace the TODO comment in the place of the generic parameter with the name of our Domain Service class, ProductsDomain. The declaration of the service class should look as follows:

public class ProductsDataService : DataService<ProductsDomain>, IServiceProvider

1. In the body of the InitializeService() method, you’ll find another TODO comment, along with some commented out lines of code that start with “config.Set…”. Remove the TODO comment and uncomment the two lines of code.
2. Change the two lines of code in InitializeService() so that the method looks as follows (replace the sample entity set and service operation names with “\*” and change the EntitySetRights for the entity set access rule to EntitySetRights.All):

public static void InitializeService(IDataServiceConfiguration config)

{

config.SetEntitySetAccessRule("**\***", EntitySetRights.**All**);

config.SetServiceOperationAccessRule("**\***", ServiceOperationRights.All);

}

This gives us an ADO.NET Data Service that exposes the data returned from our Domain Service class as a set of URI-addressable resources and maps HTTP methods against these resources to the CRUD methods we implemented on ProductsDomain. In the following steps, we’ll experiment with the data service to understand its capabilities and the data formats it uses on the wire.

1. Build and run the project.
2. In your browser, navigate to the ProductsDataService.svc.
   1. On my local web server, this ends up being <http://localhost:1361/ProductsDataService.svc/>. On your system, the host part of the URL (localhost:1361) may be different.
   2. In the following steps, I will use <http://localhost:1361/> as the local web server root – please replace this with whatever your system’s root is.
3. The document returned from <http://localhost:1361/ProductsDataService.svc/> is referred to as a Service Document and its contents are defined by the AtomPub standard. The service document for our service indicates that we have a single collection called Product, accessible via a relative URL by the same name.
4. Navigate to the Product collection at <http://localhost:1361/ProductsDataService.svc/Product>. This returns an Atom feed containing all the product data. The key things to notice about the feed are:
   1. Each product entity is represented by an <entry> element.
   2. Each <entry> element contains a <content> element with a child element called <properties>. The <properties> element contains the property values for each product entity.
5. The ADO.NET Data Services protocol defines URI syntax for navigating within a model and for performing queries. Try the following examples:
   1. <http://localhost:1361/ProductsDataService.svc/Product(1)> returns the single product with ID 1.
   2. <http://localhost:1361/ProductsDataService.svc/Product(1)/Name> returns just the Name property of the product with ID 1.
   3. <http://localhost:1361/ProductsDataService.svc/Product(1)/Name/$value> returns the raw value (without any XML adornment) of the Name property of the product with ID 1.
   4. [http://localhost:1361/ProductsDataService.svc/Product?$filter=Color eq 'Red'](http://localhost:1361/ProductsDataService.svc/Product?$filter=Color%20eq%20'Red') returns all products whose Color property has the value ‘Red’.
   5. [http://localhost:1361/ProductsDataService.svc/Product?$filter=Color eq 'Red'&$top=3](http://localhost:1361/ProductsDataService.svc/Product?$filter=Color%20eq%20'Red'&$top=3%20) returns only the top 3 products whose color is red.

In each of the queries above, ADO.NET Data Services generates a call into the GetProduct() method of our ProductsDomain class to obtain the root query and then composes additional query elements (such as filter predicates) corresponding the request URI. The URI syntax and conventions are part of the ADO.NET Data Services protocol so any client that understands this protocol can now consume data from our ProductsDomain class.

The examples we looked at in this section cover only a small part of the capabilities of the ADO.NET Data Services protocol. In particular, it’s worth noting that ADO.NET data services can serialize data using JSON (in addition to the Atom format we just saw) and supports both read and write operations (using the HTTP POST, PUT, and DELETE methods). For more information on ADO.NET Data Services capabilities, see the [Using Microsoft ADO.NET Data Services](http://msdn.microsoft.com/en-us/library/cc907912.aspx) document on MSDN.

## Using the ADO.NET Data Services Protocol from the Client

In the previous section, we added an ADO.NET Data Service to our application and explored it from the browser. But the Silverlight client part of the application does not yet communicate with this service via the ADO.NET Data Services protocol. In this section, we modify our existing Silverlight client application to use the new protocol.

Using the ADO.NET Data Services protocol requires changes to the generated ProductsDomain client class on which our Silverlight application is built. Microsoft .NET RIA Services generates the ProductsDomain client class based on the ProductsDomain DomainService class we created on the server. By marking the DomainService class on the server with a special attribute, we can instruct Microsoft .NET RIA Services to generate the client code needed to use the ADO.NET Data Services protocol. We then need to make a few changes to our Silverlight code to consume the new generated client classes. The following steps take you through this process:

1. Open the ProductsDomain.cs file in the ProductsWithDataService.Web project.
2. Add the following using declaration:

using System.Data.Services.Providers.DomainService;

1. Add the following attribute to the ProductsDomain class declaration, immediately beneath the existing [EnableClientAccess] attribute:

[DomainIdentifier("Atom", CodeProcessor = typeof(AtomCodeProcessor))]

1. In Solution Explorer, right-click on the ProductsWithDataService application (the Silverlight one – not the Web application) and choose Add Reference.
2. Click on the Browse tab, navigate to the %ProgramFiles%\Microsoft SDKs\RIA Services\v1.0\Libraries\Silverlight directory and select the following two assemblies:
   1. System.Windows.Ria.Atom.dll
   2. System.Data.Services.Client.Protocol.dll
3. Rebuild the solution. You will see some build errors because the client classes that Microsoft .NET RIA Services generated after we applied the Atom DomainIdentifier attribute are different than those against which we originally wrote our Silverlight code. We’ll fix those errors in the next two steps.
4. Open the MainPage.xaml.cs file and add the following using declarations:

using System.Windows.Browser;

using System.Data.Services.Client.Protocol;

1. Change the declaration of the \_domain member of the MainPage class to read as follows:

ProductsDomain \_domain = new ProductsDomain(

new AtomDomainClient(

new Uri(HtmlPage.Document.DocumentUri, "ProductsDataService.svc")));

1. Rebuild and run the application.

Running the application now, you’ll notice that the functionality is the same as before – the differences are all “under the hood”. With the changes we made in this section, the Silverlight client uses the ADO.NET Data Services protocol to communicate with the data service. The data service translates the HTTP requests it receives into the appropriate CRUD method calls on the ProductsDomain class.

## Summary

This document walked through the process of adding an ADO.NET Data Service to a Microsoft .NET RIA Services application and changing the client code to use the ADO.NET Data Services protocol. This illustrates the first step towards our eventual goal of these two technologies aligning on a single protocol.

As mentioned earlier in this document, the alignment work between Microsoft .NET RIA Services and ADO.NET Data Services is still at an early stage. Though we are able to get a basic application working, there are still several parts of the end-to-end Microsoft .NET RIA Services experience that are not yet implemented over the ADO.NET Data Services protocol. For example, detailed information about validation errors encountered on the server do not yet flow to the client.

In the coming months we will be making the necessary extensions to the ADO.NET Data Services protocol to fully encompass the Microsoft .NET RIA Services experience. At the end of this effort, you’ll see only one protocol and won’t need to go through many of the manual configuration steps outlined in this document.

# Breaking Changes From May 2009 CTP

This document discusses the changes made to Microsoft .NET RIA Services between the May 2009 CTP and July 2009 CTP release.

## Client Components

### DomainContext API Changes

The **DomainContext** class has changed significantly since the May 2009 CTP release.

#### Load and Cancellation APIs

The main change is in how a load method is called, how a query is formed/passed-in and how the operations are tracked through the lifecycle of invocation, callback/event handler registration and completion/cancellation. All three of the primary operation types (Query, Submit, and Invoke) now share a unified eventing/callback/cancellation model. See ”Asynchronous Domain Operations” for feature description.

##### Previously

**Authored code**

In server project

|  |
| --- |
| [EnableClientAccess]  public class HRSystem : LinqToSqlDomainService<HRDataContext>  {  public IQueryable<Employee> GetReports(int managerID)  {  // ...  }  } |

**Generated code**

In client project

|  |
| --- |
| public sealed partial class HRSystem : DomainContext  {      public static IQueryable<Employee> EmployeesQuery { … }      public EntityList<Employee> Employees { get; }      public void LoadReports(  int managerID,  IQueryable<Employee> query,  MergeOption mergeOption,  object userState)  {  // ...  }      public void LoadReports(managerID, IQueryable<Employee> query, object userState)  {  // ...  }      public void LoadReports(int managerID)  {  // ...  }  } |

**Usage Pattern**

Client code-behind

* It was necessary to subscribe to DomainContext Loaded and Submitted events and did your work in the handler, possibly using user state to identify particular requests.
* ServiceOperations included a completion event in addition to the generated method.

|  |
| --- |
| public partial class EmployeeList : Page  {      HRSystem hr = new HRSystem();      public EmployeeList()      {          this.InitializeComponent();          this.dataGrid1.ItemsSource = this.hr.Employees;          this.hr.Loaded += this.hr\_Loaded;            // Query to be sent to server tier          IQueryable<Employee> contractorQuery =  from e in HRSystem.EmployeesQuery              where e.SalariedFlag == false              select e;          // Some userState to identify the call          int LoadEmployeeToken = 1;          hr.LoadEmployeesByManager(  mgrID,  contractorsQuery,  LoadEmployeeToken);      }      private void hr\_Loaded(object sender, LoadedDataEventArgs e)      {  // ...      }  // ...      private void CancelButtonClicked(object sender, EventArgs e)      {  // Elsewhere – say in cancel button click handler  // Same userState to identify the pending operation to be cancelled      this.hr.CancelRequests(LoadEmployeeToken);  }  } |

##### New APIs/Behavior

**Authored code**

In server project

*Same as before – no change*

**Generated code**

In client project

|  |
| --- |
| public class HRSystem : DomainContext  {  public EntityList<Employee> Employees { get; }  public EntityQuery<Employee> GetReportsQuery(int managerID)      {  // ...      }  } |

**Usage Pattern**

Client code-behind

* Generated **DomainContext** proxies now have a single **Load** method that accepts **EntityQuery**<TEntity> queries.
* You can subscribe to the **Completed** event directly on the on the **LoadOperation** and **SubmitOperation** instances.
* The **ServiceOperation** generated event goes away, and the method now returns an **InvokeOperation** whose **Completed** event can be used as shown below.

|  |
| --- |
| public partial class EmployeeList : Page  {      HRSystem hr = new HRSystem();  LoadOperation<Employee> contractorsLoadOperation;      public EmployeeList()      {          this.InitializeComponent();          this.dataGrid1.ItemsSource = this.hr.Employees;          // Different query root with better compile-time error experience  // for unsupported LINQ operators.          EntityQuery<Employee> contractorsQuery =  from e in hr.GetReportsQuery(managerID)              where e.SalariedFlag == false              select e;          contractorsLoadOp = hr.Load(contractorsQuery);    // You can either subscribe to the Completed event for completion.  // Alternatively, you can also pass in a callback into the above  // Load method rather than the Completed event.  contractorsLoadOp.Completed += OnContractorsLoaded;      }      private void OnContractorsLoaded(object sender, EventArgs e)      {  LoadOperation<Employee> contractorsLoadResult =  (LoadOperation<Employee>)sender;  foreach (Employee employee in contractorsLoadResult.Entities)  {  // ...          }      }  // ...      private void CancelButtonClicked(object sender, EventArgs e)      {  // Elsewhere – say in cancel button click handler  contractorsLoadOp.Cancel();  }  } |

**API Outline**

Only relevant API changes are shown below

|  |
| --- |
| public abstract class DomainContext : INotifyPropertyChanged  {  // Invoke Methods  public InvokeOperation InvokeServiceOperation(  string operationName,   IDictionary<string, object> parameters,   delegate callback,   object userState);  public virtual InvokeOperation InvokeServiceOperation(  string operationName,   Type returnType,   IDictionary<string, object> parameters,   delegate callback,   object userState);  // Load Methods  public LoadOperation<TEntity> Load<TEntity>(  EntityQuery<TEntity> query,   MergeOption mergeOption,   Action<LoadOperation<TEntity>> callback,   object userState) where TEntity : Entity;  public LoadOperation<TEntity> Load<TEntity>(  EntityQuery<TEntity> query,   Action<LoadOperation<TEntity>> callback,   object userState) where TEntity : Entity;  public LoadOperation<TEntity> Load<TEntity>(  EntityQuery<TEntity> query) where TEntity : Entity;  public LoadOperation<TEntity> Load<TEntity>(  EntityQuery<TEntity> query,   MergeOption mergeOption) where TEntity : Entity;  public virtual LoadOperation Load(  EntityQuery query,   MergeOption mergeOption,   delegate callback,   object userState);  // Persistence Methods  public void RejectChanges();  public SubmitOperation SubmitChanges();  public virtual SubmitOperation SubmitChanges(  Action<SubmitOperation> callback, object userState);  }  // Used with loading operations  public sealed class LoadOperation<TEntity> : LoadOperation where TEntity: Entity  {  // Properties  public IEnumerable<TEntity> Entities { get; }  public EntityQuery<TEntity> EntityQuery { get; }  }  // Used with invocation operations  public sealed class InvokeOperation<TValue> : InvokeOperation  {  // Properties  public TValue Value { get; }  }  // Used with submit operations  public sealed class SubmitOperation : OperationBase  {  // Properties  public EntityChangeSet ChangeSet { get; }  public IEnumerable<Entity> EntitiesInError { get; }  protected override bool SupportsCancellation { get; }  } |

##### Reasons for Change

* In the last CTP, a **DomainService** query operation named ‘GetEmployees’ would be named ‘LoadEmployees’ in the generated client proxy. Now we generate a method for each query that returns an **EntityQuery**<TEntity> that can be used with the new **DomainContext**.**Load** method. (See above example.)
* Better model for forming and passing in LINQ queries – better discoverability and intuitive
* Better compile-time type checking with a type that exposes supported subset of standard LINQ query operators
* Better pattern for specifying callbacks / event handlers when a given operation is completed
* Simpler way to consume results of a single load operation without accumulating results (the accumulated results usage pattern remains unchanged)
* Clear cancellation capability that can cancel a specific operation
* A model that works across queries, service operations and **SubmitChanges**()
* Enable a clean model for multiple concurrent asynchronous operations.

##### New APIs

* The **EntityQuery**, **EntityQuery**<T> types have been introduced to represent entity queries.
* **OperationBase**, **LoadOperation**, **SubmitOperation** and **InvokeOperation** have been introduced to represent asynchronous operations.
  + All operations implement **INotifyPropertyChanged**.
* The **DomainContext**.**Submit**() signature has changed as shown above.
* **DomainContext**.**Submit** and **DomainContext**.**Load** are now virtual methods.
* A \*Query method will now be generated in the **DomainContext** proxy type for each query method exposed in a **DomainService**.

##### Deprecated APIs

* **DomainContext** no longer implements **IDomainContext**.
* **IDomainContext** has been removed.
* Cancellation APIs have been removed (**CancelSubmit**, **CancelAllRequests**), moved into the operation types shown above.
* Loading & Submitting and service operation events have been removed; equivalent events have been moved into the operation types shown above.
* **LoadedDataResults**, **SubmittedChangesResults** have been removed.
* **LoadMethodAttribute** has been removed.
* Static query root properties have been removed, replaces with \*Query methods shown above.
* *Related*: **QueryAttribute**.**PreserveName** property has been removed.

#### Cross-DomainContext API Changes

The cross-DomainContext APIs have changed.

#### Previously

Previously, one could add a reference to a DomainContext instance and all entity types contained in that DomainContext were eligible for cross-DomainContext lookup.

|  |
| --- |
| public void InitializeDomainContexts()  {  // Instantiate our providers  MyOrdersContext myOrders = new MyOrdersContext();  ProductCatalogContext productCatalog = new ProductCatalogContext();  // Add 'productCatalog' as a reference for external lookups.  myOrders.AddReference(productCatalog);  // Start loading data in each DomainContext so that the  // cross-DomainContext associations will be populated.  // ...  } |

##### New APIs/Behaviors

The AddReference API has changed so that a specific entity type must be specified when adding a reference.

|  |
| --- |
| public void InitializeDomainContexts()  {  // Instantiate our providers  MyOrdersContext myOrders = new MyOrdersContext();  ProductCatalogContext productCatalog = new ProductCatalogContext();  // Add 'productCatalog' as a reference for external lookups for  // the 'Product' type specifically.  myOrders.AddReference<Product>(productCatalog);  // Start loading data in each DomainContext so that the  // cross-DomainContext associations will be populated.  // ...  } |

##### Related Changes

The equivalent cross-DomainContext APIs on EntityContainer have changed to match the changes shown above.

### DomainDataSource API Changes

See “Using the Silverlight DomainDataSource” for description of the new API.

The following query method in a DomainService type will be referenced in the subsequent API change examples:

|  |
| --- |
| [Query]  public IQueryable<City> GetCities()  {  this.\_cityData.Cities.AsQueryable<City>();  } |

##### Previously

|  |
| --- |
| myDomainDataSource.LoadMethodName = "LoadCities";  myDomainDataSource.LoadParameters =  new System.Windows.Data.ParameterCollection(); |

##### New APIs/Behaviors

|  |
| --- |
| myDomainDataSource.QueryName = "GetCities";  myDomainDataSource.QueryParameters =  new System.Windows.Data.ParameterCollection(); |

##### Deprecated APIs

* The **LoadMethodName** property has been removed; a new property **QueryName** has been added as a replacement.
  + The value provided for **QueryName** must match the name of the query operation in the targeted ***DomainService***.
* The **LoadParameters** property has been removed; a new property **QueryParameters** has been added as a replacement.

##### Other Changes

##### **EventArg Namespace Changes**

* The following **EventArg** types were previously defined within the *System.Windows.Ria* assembly under the *System.Windows.Ria.Data* namespace:
  + **LoadedDataEventArgs**
  + **SubmittingDataEventArgs**
  + **SubmittedDataEventArgs**
* These classes have been moved to the *System.Windows.Ria.Controls* assembly under the *System.Windows.Controls* namespace.

##### **LoadingDataEventArgs API Changes**

* Previously exposed an **IQueryable** Query property for manipulating the query within the **LoadingData** event;
* Now exposes an **EntityQuery** Query property for manipulating the query within the **LoadedData** event;
* The **Query** must be cast to an **EntityQuery**<T> to gain the extension method operators: OrderBy, OrderByDescending, Skip, Take, ThenBy, ThenByDescending, and Where.

##### **SubmittedChangedEventArgs API Changes**

* This class previously derived from **CancelEventArgs**, but it now derives directly from **EventArgs**;
* To cancel a submission, use **DomainDataSource**.**CancelSubmit**().

### Application Services API Changes

#### General

The client Application Service DomainContext types have changed to coincide with the DomainContext changes described above.

#### Setup

In an effort to make application services easier to consume, we replaced the *UserService* with *RiaContext*. This impacts initialization in *App.xaml*:

##### Previously

|  |
| --- |
| <Application.Services>  <appsvc:WebUserService x:Name="UserService" />  </Application.Services> |

##### New APIs/Behaviors

Note that we are making use of ‘Application.ApplicationLifetimeObjects’ which is new to Silverlight 3.

|  |
| --- |
| <Application.ApplicationLifetimeObjects>  <app:RiaContext>  <app:RiaContext.Authentication>  <appsvc:FormsAuthentication/>  </app:RiaContext.Authentication>  </app:RiaContext>  </Application.ApplicationLifetimeObjects> |

#### Accessing UserService

This also impacts static access to the *UserService* and User in your code.

##### Previously

|  |
| --- |
| UserService.Current.Login("username", "password");  string email = ((MyUser)UserService.Current.User).Email; |

##### New APIs/Behaviors

|  |
| --- |
| RiaContext.Current.Authentication.Login("username", "password");  string email = RiaContext.Current.User.Email; |

Something notable is that *RiaContext* is generated into your application’s default namespace. This allows strongly-typed access to all user properties. Alternatively, if you do not define a server-side authentication service, the *User* property will not be generated.

#### Client-side authentication

*UserService* was renamed to *AuthenticationService* as part of this refactor and has undergone significant revision. Most of the API is analogous but is now exposed using the operation types introduced by *DomainContext*. Notably, *Cancel* has been removed in favor of the new pattern and can be found on the operation.

##### Previously

|  |
| --- |
| public event EventHandler<LoginCompletedEventArgs> LoginCompleted;  public abstract void Login(LoginParameters parameters); |

##### New APIs/Behaviors

|  |
| --- |
| public bool IsLoggingIn { get; }  public LoginOperation Login(LoginParameters parameters);  public LoginOperation Login(  LoginParameters parameters,  Action<LoginOperation> completeAction,  object userState); |

#### Binding

One significant addition is *RiaContext* and *AuthenticationService* both support *INotifyPropertyChanged*. They can now be easily data-bound to in XAML. To facilitate this currently, you’ll need to add a single line to your startup handler in *App.xaml.cs*.

|  |
| --- |
| private void Application\_Startup(object sender, StartupEventArgs e)  {  this.Resources.Add("RiaContext", RiaContext.Current);  this.RootVisual = new MainPage();  } |

Then in your XAML you can do the following:

|  |
| --- |
| <!-- Binding to IsBusy -->  <Button x:Name="Login" Content="Login" Click="Login\_Click"  DataContext="{StaticResource RiaContext}"  IsEnabled="{Binding Authentication.IsBusy, Converter={StaticResource Inverter}}"/>  <!-- Binding to IsAuthenticated -->  <TextBlock DataContext="{StaticResource RiaContext}"  Text="{Binding User.IsAuthenticated}"/> |

## Server Components

### DomainService Changes

#### DomainService Requirements

##### Sharing of Entity Types Across DomainServices

Previously, we allowed for sharing of entity types across multiple **DomainService** types. (Example: exposing a “Product” entity in “ProductsCatalogDomainService” and “ProductManagerDomainService”.) Sharing of entity types across **DomainService** types will now generate a build error.

##### Nested DomainService Types

Nested **DomainService** types are no longer supported.

#### Service Operation Changes

Service operations are no longer allowed to return entities. Query operations must be used for that purpose.

#### Update Method Support

See “How to modify entities” for the current description of the feature.

Update methods accepting 2 parameters (current and original) are no longer supported. Update methods accepting 1 parameter will be passed the *current* entity value.

|  |
| --- |
| public void UpdateProduct(Product current) { ... }  ~~public void UpdateProduct(Product current, Product original) { ... }~~ |

### Delete Method Support

Delete methods accepting 2 parameters (current and original) are no longer supported. Delete methods accepting 1 parameter will be passed the *original* entity value.

|  |
| --- |
| public void DeleteProduct(Product original) { ... }  ~~public void DeleteProduct(Product current, Product original) { ... }~~ |

#### DomainService API Changes

In addition to these changes, a **ChangeSet** property has been added to the **DomainService** type. This property will be set before performing DomainService submission requests such as insert, update, delete or custom operations. This property will be null for other request types.

|  |
| --- |
| public class DomainService  {     // New property added     protected ChangeSet ChangeSet { get; }     // ...  } |

#### ChangeSet API Changes

The following methods have been added to the **ChangeSet** type:

* **AssociateWith**<TEntity> (signature shown below)

This method can be used to associate a server-side entity with a client-modified entity. When used, subsequent operations will use the associated entity in place of the entity provided by the client. This can be used to perform masked-updates on the server.

* **GetOriginal**<TEntity> (signature shown below)

This method will return the original entity (as seen by the client) when provided with a client-modified entity.

|  |
| --- |
| public class ChangeSet  {     // Associates serverModifiedEntity with clientModifiedEntity     // The latter is sent back to client if SubmitChanges() succeeds     // Throws ArgumentException if clientModifiedEntity is not in ChangeSet     public void AssociateWith<TEntity>(  TEntity clientModifiedEntity,  TEntity serverModifiedEntity) { ... }     // Returns original version for the given entity     // Throws ArgumentException if clientModifiedEntity is not in ChangeSet     public TEntity GetOriginal<TEntity>(TEntity clientModifiedEntity) { ... }  } |

### Entity Framework and LINQ-to-SQL DomainServices

##### Previously

The TypeDescriptionProviders for Entity Framework and LINQ-to SQL infer the RequiredAttribute attribute if the DAL metadata indicates that the entity member is a nullable type but mapped to a non-nullable column in the database. The attribute was enforced in Silverlight in a way that precludes null value (intended and consistent with database semantics) and empty string (not consistent with database semantics). This dual enforcement is consistent with the semantics used by ASP.NET Dynamic Data in 3.5 SP1 but not consistent with the database semantics. Currently, the scenarios where the database column is non-nullable but empty strings are allowed cannot be implemented without changing generated code.

The generated code for a string member ‘Title’ mapped to a non-nullable column would look like:

|  |
| --- |
| [Required]  [StringLength(100)]  public string Subtitle |

##### New APIs/Behaviors

The breaking change is to not infer the RequiredAttribute attribute on string members and require users to put it in manually if they want to get the current behavior. This will break existing apps that rely on automatic generation and resulting enforcement of preventing null and empty strings. The app authors will have to manually add the metadata attribute in buddy class or other source of metadata as applicable.

Now the generated code for a string member ‘Title’ mapped to a non-nullable column looks like:

|  |
| --- |
| [StringLength(100)]  public string Subtitle |

### SilverlightApplication and SeoSilverlightApplication Control Changes

The **SilverlightApplication** (*System.Web.Ria* assembly) and **SeoSilverlightApplication** control (*System.Web.DomainServices.Controls* assembly) have been removed.

##### Reason for Change

Both the **SilverlightApplication** and **SeoSilverlightApplication** controls derived from the ASP.NET **Silverlight** control which has since been removed from the Silverlight SDK. It is now recommended to hand-craft your own HTML <OBJECT> markup or use the Silverlight.js file as recommended for general Silverlight projects.

(Note that using JavaScript rendering precludes you from having <OBJECT> tag alternate content indexed by search engines. For more information on SEO and <OBJECT> tags, see the whitepaper posted here: <http://silverlight.net/learn/whitepapers/seo.aspx>.)

### Removal of SharedAttribute Used in Code Generation

The existing **SharedAttribute** custom attribute from has been removed.  It lived in the *System.Web.Ria.Data* namespace in the *System.Web.Ria.dll* assembly.

This attribute was formerly used to tell the client proxy code generator which types and members were already visible on the client project.  With the removal of this attribute, the code generator has been updated to detect shared types and members automatically.

##### Previously

In the May 09 Preview, the use of the **SharedAttribute** attribute would have looked like the following.  It could be used to mark both types as well as members.

|  |
| --- |
| namespace Purchasing  {  **[Shared]**  public class CreditCardValidator  {  public static ValidationResult ValidateCard(  CreditCard creditCard, ValidationContext context)  {  // ...  }  }  } |

##### New APIs/Behaviors

To make this code work now, it is only necessary to delete the **SharedAttribute** attribute:

|  |
| --- |
| namespace Purchasing  {  //[Shared] <-- removed  public class CreditCardValidator  {  public static ValidationResult ValidateCard(  CreditCard creditCard, ValidationContext context)  {  // ...  }  }  } |

##### Related Changes

To detect shared types and members, the code generator now requires a PDB file for any assembly from a project that might contain shared code.  This lets the code generator determine whether types and members it encounters during code generation will also be present on the client.

By default, PDB’s are generated even for Release builds, so no action is required for most scenarios.  If you modified your project to prevent the PDB from being built, you will encounter a code generation warning that the PDB cannot be located and describing how to rectify that problem.

Also, you may now find more attributes appearing in the generated client proxy classes.  The metadata pipeline has been modified to generate an equivalent custom attribute declaration for every custom attribute it encounters on the server-side if it can determine it is also visible on the client.  Prior to this change, only attributes marked with **SharedAttribute** (or a known set of .NET RIA Services attributes) were propagated into the proxy classes.  Now, these attributes will be propagated if the code generator can determine those types exist on the client.

## Data Annotations

### Validator Class Changes

##### Previously

**Validator.TryValidateObject** and **Validator.ValidateObject** would validate entity-level validation attributes regardless of the success or failure of the property-level validation attributes.

##### New APIs/Behavior

**Validator.TryValidateObject** and **Validator.ValidateObject** will only validate entity-level validation attributes if all property-level attributes were validated successfully.

##### Reason for Change

With the introduction of **IValidatableObject**, we only want to call **IValidatableObject.Validate** if previous validations were successful.  Therefore we check for errors on property attributes and entity attributes before proceeding to the interface.  However, we want behavior between the entity attributes and the interface to be consistent—if the property attributes failed, then we don't want to invoke the entity attributes.

1. You can also add a Silverlight application to an existing web application project using Add New Project. [↑](#footnote-ref-1)
2. When “Enable Editing” is checked, the DomainService class will expose methods to create, update, and delete the corresponding entity. [↑](#footnote-ref-2)
3. The .NET RIA Services framework has base classes for the Entity Framework and LINQ to SQL. It also has a technology-agnostic DomainService base class. [↑](#footnote-ref-3)
4. Using the “Enable .NET RIA Services” checkbox during project creation [↑](#footnote-ref-4)
5. Multiple Silverlight client projects can be associated with a single ASP.NET web application, and each will receive its own client proxy classes. [↑](#footnote-ref-5)
6. Specifically, any file named \*.shared.cs or \*.shared.vb is assumed to be shared between tiers [↑](#footnote-ref-6)
7. Except for properties whose type would not be available on the Silverlight client, such as DAL-specific types [↑](#footnote-ref-7)
8. This merge is technically done at a level below code generation. Even the server-side entity merges these attributes when performing server-side validation. [↑](#footnote-ref-8)
9. A query method is any method that returns a collection of entities, generally as IEnumerable<T> or IQueryable<T>. [↑](#footnote-ref-9)
10. Note that Entity type members exposed to the client are also restricted to the list of Supported Types [↑](#footnote-ref-10)
11. Based on documentation on .NET Framework primitive types [here](http://msdn.microsoft.com/en-us/library/system.type.isprimitive.aspx) [↑](#footnote-ref-11)
12. Note that System.Data.Linq.Binary types are exposed on the client as Byte[] [↑](#footnote-ref-12)
13. This option is known as the “RIA Link” and identifies a project to use as the source for the proxy types. [↑](#footnote-ref-13)
14. Entity types are “exposed” by a DomainService simply by appearing in the query method signatures. [↑](#footnote-ref-14)
15. The RIA Link is shorthand for the “Enable .NET RIA Services” option in the project properties of Silverlight projects [↑](#footnote-ref-15)
16. Think about setting breakpoints or single-stepping code that runs on multiple tiers and in multiple apps concurrently. Having separate physical copies of the files at runtime makes this experience easier to manage. [↑](#footnote-ref-16)
17. This is short-hand for the “Enable .NET RIA Services” option available whenever a Silverlight application is created. [↑](#footnote-ref-17)
18. The “Enable .NET RIA Services” checkbox was checked when this solution was created. [↑](#footnote-ref-18)
19. The solution folder is not required but helps keep the 2 parts together within the solution. [↑](#footnote-ref-19)
20. You need to activate “Show All Files” on the client library project to see the generated code files. [↑](#footnote-ref-20)
21. You also need to add a reference to System.Windows.Ria to use .NET RIA Services types. [↑](#footnote-ref-21)
22. The “Enable .NET RIA Services” option was checked in each Silverlight application project [↑](#footnote-ref-22)
23. This is arguably a bug because project-to-project references could be used to locate the files. But if conventional assembly references are used, the shared source code cannot be located. [↑](#footnote-ref-23)
24. Or examining Project Properties of an existing Silverlight project [↑](#footnote-ref-24)