

**WCF RIA Services**

**WCF RIA Services**

**Author: Brian Noyes**

**June 2011**

**Published by SilverlightShow**

**©2011 SilverlightShow**

Table of Contents

[Chapter 1: Getting Started 6](#_Toc297040417)

[Introduction 6](#_Toc297040418)

[Step 1: Create the Silverlight Application Project and Link to the Server Project 7](#_Toc297040419)

[Step 2: Create the Domain Model Entities 9](#_Toc297040420)

[Step 3: Define a Domain Service 11](#_Toc297040421)

[Step 4: Retrieve Data into the UI with DomainDataSource 13](#_Toc297040422)

[Summary 14](#_Toc297040423)

[Chapter 2: Querying Data 15](#_Toc297040424)

[Overview 15](#_Toc297040425)

[Step 1: Add a parameterized collection query to your domain service 15](#_Toc297040426)

[Step 2: Add UI to execute the query 15](#_Toc297040427)

[Step 3: Execute a query on the client through the DomainContext 16](#_Toc297040428)

[Step 4: Add query methods that return a single entity 17](#_Toc297040429)

[Convention Over Configuration 17](#_Toc297040430)

[Custom Domain Services 18](#_Toc297040431)

[Summary 19](#_Toc297040432)

[Chapter 3: Updating Data 20](#_Toc297040433)

[Overview 20](#_Toc297040434)

[IQueryable<T> and the Magic of Expression Trees 20](#_Toc297040435)

[DomainContext Caching and Change Tracking 21](#_Toc297040436)

[Step 1: Add update methods to your domain service 21](#_Toc297040437)

[Step 2: Add some UI to invoke an add operation 22](#_Toc297040438)

[Step 3: Create a new Task, add it to the domain context, and submit changes 23](#_Toc297040439)

[DomainContext’s Async API 23](#_Toc297040440)

[Summary 24](#_Toc297040441)

[Chapter 4: Integrating with the Model-View-ViewModel Pattern 25](#_Toc297040442)

[Introduction 25](#_Toc297040443)

[Step 1: Factor out a view model 25](#_Toc297040444)

[Step 2: Hook up the view to the view model 28](#_Toc297040445)

[Summary 28](#_Toc297040446)

[Chapter 5: Metadata and Shared Classes 30](#_Toc297040447)

[Introduction 30](#_Toc297040448)

[Metadata Classes 30](#_Toc297040449)

[Shared Code 33](#_Toc297040450)

[Summary 34](#_Toc297040451)

[Chapter 6: Validating Data 35](#_Toc297040452)

[Introduction 35](#_Toc297040453)

[Data Annotation Validation Attributes 35](#_Toc297040454)

[Custom Validation Attributes 37](#_Toc297040455)

[Server-Side Validation 39](#_Toc297040456)

[Async Validation through Invoke Methods 40](#_Toc297040457)

[Summary 41](#_Toc297040458)

[Chapter 7: Authentication and Authorization 42](#_Toc297040459)

[Introduction 42](#_Toc297040460)

[Authenticating the User 43](#_Toc297040461)

[Authorization with RIA Services 48](#_Toc297040462)

[Requiring Message Protection 50](#_Toc297040463)

[Summary 50](#_Toc297040464)

[Chapter 8: Testing and Debugging 52](#_Toc297040465)

[Introduction 52](#_Toc297040466)

[Unit Testability 52](#_Toc297040467)

[Unit Testing the View Model or Client-Side Service That Uses a DomainContext 53](#_Toc297040468)

[Creating a Mock DomainClient 55](#_Toc297040469)

[Testing a View Model Command Handler for a Query 56](#_Toc297040470)

[Testing a View Model Command Handler for SaveChanges 59](#_Toc297040471)

[Unit Testing Domain Service Code 61](#_Toc297040472)

[Debugging RIA Services 62](#_Toc297040473)

[Summary 64](#_Toc297040474)

[Chapter 9: Structuring Your Application 65](#_Toc297040475)

[Introduction 65](#_Toc297040476)

[Client-Server Project Relationships 65](#_Toc297040477)

[Adding a WCF RIA Services Class Library Project 66](#_Toc297040478)

[Using Entities across Domain Services 68](#_Toc297040479)

[Breaking Your Client Application into Multiple Modules 68](#_Toc297040480)

[Using MEF To Dynamically Load the Module 71](#_Toc297040481)

[Summary 72](#_Toc297040482)

[Chapter 10: Exposing Domain Services To Other Clients 73](#_Toc297040483)

[Introduction 73](#_Toc297040484)

[Exposing an OData Endpoint From Your Domain Service 73](#_Toc297040485)

[Consuming the OData Endpoint From a .NET Client 75](#_Toc297040486)

[Exposing a SOAP Endpoint From Your Domain Service 76](#_Toc297040487)

[Consuming the SOAP Endpoint From a .NET Client 77](#_Toc297040488)

[Exposing a REST/JSON Endpoint 77](#_Toc297040489)

[Summary 78](#_Toc297040490)

[About the Author 79](#_Toc297040491)

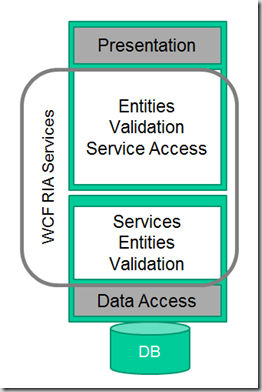
# Chapter 1: Getting Started

### Introduction

In order to build serious business application in Silverlight (and other client technologies), you have to work with a lot of data. And that data is usually not resident on the client machine, it is usually distributed amongst many clients and is stored and operated on by back-end services. If you try to write this kind of application architecture yourself, you have to tackle a lot of technologies and write a lot of plumbing. In the end, most of what you are doing is pushing and pulling data from the client to the back end and invoking operations on the server from the Silverlight client application. What would be great is if most of that plumbing and push-pull logic could be automated for you, allowing you to just focus on what data you need, the rules that surround the manipulation of that data, and how to present it in the client application.

This is exactly what WCF RIA Services does for you. WCF RIA Services is a new part of the .NET 4 and Silverlight 4 frameworks that lets you quickly build N-Tier Silverlight client applications without needing to focus on the service plumbing to get data into and out of your client application from back-end services, logic and data access. This book will get you up and running with all the capabilities of WCF RIA Services.

RIA Services helps you to write one set of server code, but have appropriate parts of that service code available on the client without having to duplicate it or write client side code to access it. On the server side, WCF RIA Services helps you define your services, domain entities, and supporting logic. On the client side, WCF RIA Services code generates corresponding classes that let you easily call those services, have the same entities available and populated on the client side, along with supporting validation logic and other kinds of code that you can share between the client and the service side. The diagram below shows what part of your architecture WCF RIA Services focuses on.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/6-9-2010%205-56-42%20PM_2.png)

To see this in action, in this chapter I’ll do a quick run through of the basics of WCF RIA Services. In subsequent chapters in this ebook, I’ll dive into more details of what is going on under the covers and how to adapt from the simplest usage of WCF RIA Services I’ll show in this chapter, to more real world scenarios. In those later chapters I’ll dive into things like security, validation, concurrency, customization, and combining WCF RIA Services with other practices such as the Model-View-ViewModel (MVVM) pattern and unit testing.

But first we need to cover the basics. The application I’ll be using in this ebook is a task and time management application. Imagine you need to be able to enter tasks for individuals, track the time they spend on those tasks, link the tasks to customers and projects and so on. Obviously there are a lot of other systems and applications that support this kind of functionality, but this will make a good, easy to understand domain for the application we will build up.

In this chapter we will step through the following key concepts in WCF RIA Services:

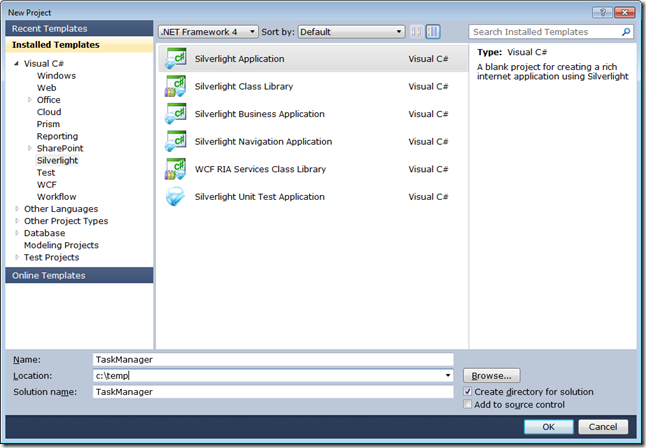
* Project Links – A link between a Silverlight client project and a server web application or class library. The server project defines or references the domain services, entities, metadata, and shared code defined on the server side. As a result of the link, appropriate code is generated in the client project at compile time.
* Domain Services – This is the core construct of WCF RIA Services. A domain service defines the operations supported on the server side. These are usually mostly focused on CRUD operations against entity types, but can also be arbitrary operations to be invoked on the server from the client. These operations are exposed automatically via WCF and can be called from the client generated code without needing to know much at all about WCF.
* Entities – you define entity types on the server, and a client-side definition of the same entity type is generated for use on the client side. You can add validation and other kinds of metadata to the entity definition which will affect the code generated on the client side, allowing you to just maintain a single server-side definition. The entity types on the client and server side are used to serialize and deserialize the data across the WCF service. The entity types can be created with Entity Framework, LINQ to SQL, or can be Plain Old CLR Objects (POCOs) that you define yourself.
* Domain Context – This is the client side counterpart to the domain service. It is generated code on the client side that gives you easy access to the functionality that resides on the server side. Internally it contains a WCF proxy that makes the service calls, and it also manages creating the queries that get executed on the server side, the change tracking for the entities being manipulated on the client and more.
* DomainDataSource – This is a data source object that can be used to execute queries and submit changes to the server side. It gets associated with the domain context and makes calls through that domain context on your behalf to query and update entities. It facilitates direct data binding from XAML or can be used behind the scenes as well.

[Sample Code](source_code/TaskManagerPart1.zip)

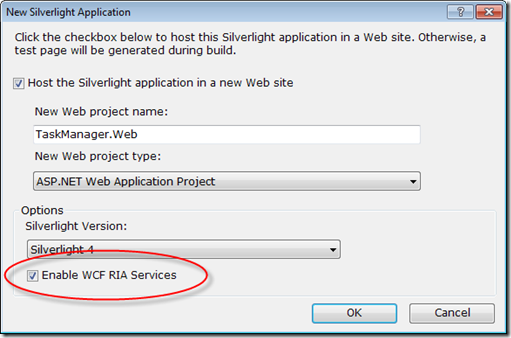
### Step 1: Create the Silverlight Application Project and Link to the Server Project

The application I’ll build in this ebook is based on a simple database that stores task and related entity information. To follow along with the steps, you will need to run the TaskManager.sql script in the download sample to create the database schema and populate it with a few sample entities. Once you have the database in place, you are ready to start creating the application. You’ll also need to have the Silverlight 4 Tools installed for Visual Studio 2010, which installs WCF RIA Services support as well.

Create a new Silverlight Application project named TasksManager. You can use WCF RIA Services with any of the Silverlight project types, including the Silverlight Business Application template that is added by WCF RIA Services itself. But to start simple, I’ll go with just the Silverlight Application template.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/NewProject_2.png)

After you click OK in the new project window, you will be presented with the normal dialog to create a hosting web application project. What you will see that is new is a checkbox at the bottom to Enable WCF RIA Services. Check that box to create the link between the client project and the server project so that WCF RIA Services can code generate the appropriate client code based on the domain services and entities you will create.

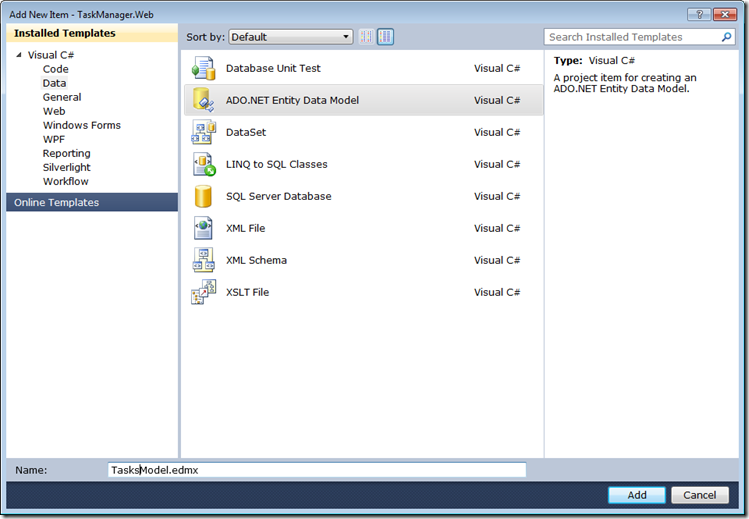
[](http://www.silverlightshow.net/Storage/Users/brian.noyes/ProjectLink_2.png)

Click OK and the projects will be created.

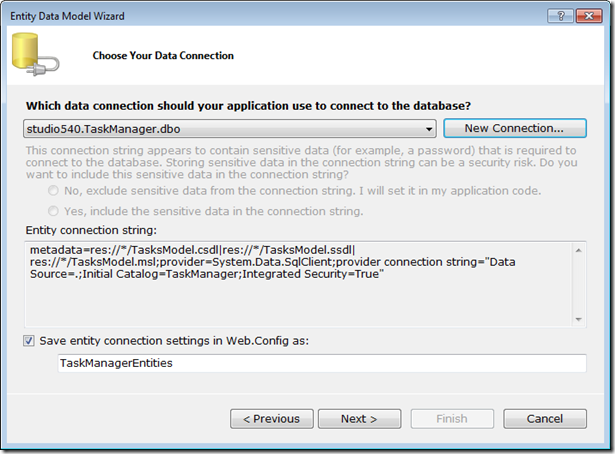
### Step 2: Create the Domain Model Entities

WCF RIA Services is mostly about moving data back and forth between the client and server for you. So you need some data to work with. Out of the box, Entity Framework is supported best. With the WCF RIA Services Toolkit, there is also support for LINQ to SQL. And you can create your own domain services if you want to work with POCOs. For this chapter, you will use Entity Framework.

Right click on the TaskManager.Web server project in Solution Explorer and select Add > New Item, pick the Data category on the left, and select ADO.NET Entity Data Model, and name it TasksModel.edmx.

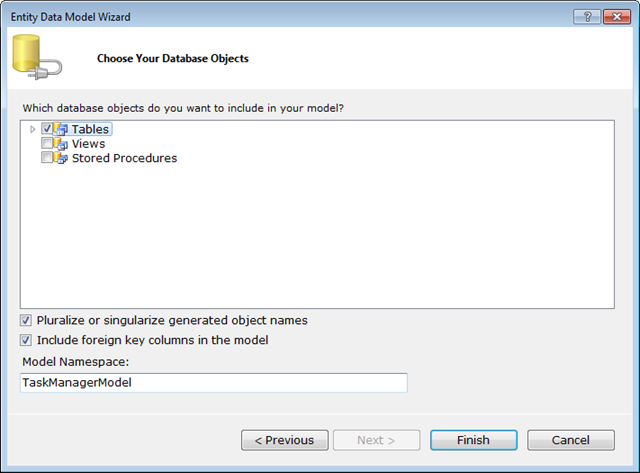
[](http://www.silverlightshow.net/Storage/Users/brian.noyes/AddEFModel_2.png)

Click Add and you will see the Entity Data Model Wizard. In the first step, leave the option to Generate from database selected and click Next. Create a New Connection in the next step to the TaskManager database you created with the sample code SQL script at the beginning.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/DataConnection_2.png)

The connection name should default to TaskManagerEntities, which is fine. Click Next.

In the next step, Choose Your Database Objects, check the box for Tables to select all the tables and click Finish.

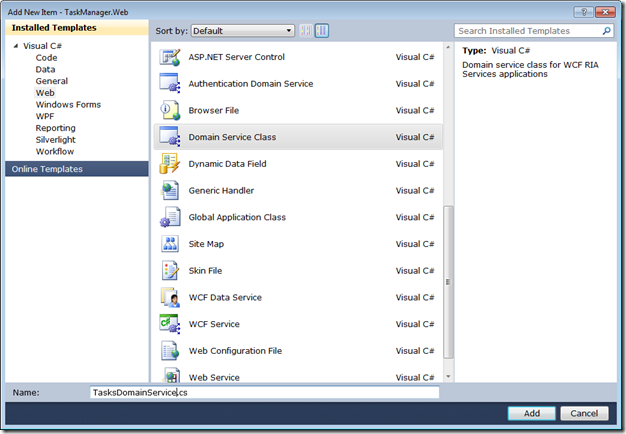


You now have an Entity Framework data model for all the tables in the database. In this chapter, you will only be working with the Task table, but in future chapters in the ebook you will work with one or two other types.

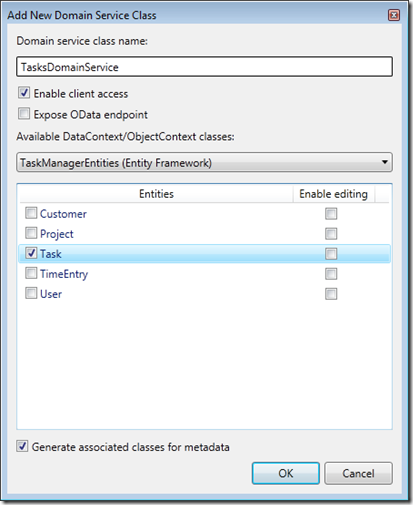
Build the solution to make sure all is well. If you forget to build after adding your entity data model, it will not show up when creating the domain service in the next step.

### Step 3: Define a Domain Service

So far, the only thing specific to WCF RIA Services you have done is check the box to create the link between the client and server project. Now you will do the core activity around RIA Services: define a domain service. Right click on the TaskManager.Web project and select Add > New Item. Select the Web category, and select Domain Service Class. Name the class TasksDomainService.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/AddDomainService_2.png)

Next you will be presented with a dialog that lets you select a number of options for your service, including the most important option of what entities it will expose.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/DomainServiceOptions_2.png)

For this chapter, you will just be working with Tasks, so select that entity type and leave the rest of the defaults selected. I’ll be getting into more detail on some of the other options and what they do in other chapters. Click OK and your domain service will be created.

Minus a bunch of comments they insert, the resulting class looks like this:

[EnableClientAccess()]

public class TasksDomainService : LinqToEntitiesDomainService<TaskManagerEntities>

{

public IQueryable<Task> GetTasks()

{

return this.ObjectContext.Tasks;

}

}

The LinqToEntitiesDomainService<T> class provides the glue between an entity data model and the services that will be exposed by WCF RIA Services. The EnableClientAccess attribute is what causes the client side code to be generated at compile time in the linked client project. From there, you add methods to your domain service to perform CRUD operations on the entities and possibly just expose operations that can be invoked from the client as well.

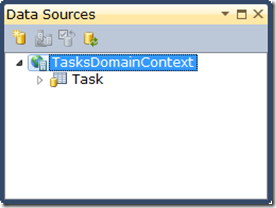
In this case, because you did not select the Enable Editing option in the previous dialog, the initial code only contains a single query method. I’ll get into more detail on the options and variations you have for query methods in the next chapter. But the one generated here uses convention to indicate that it is a query method based on the return type of IQueryable<T> and the method name. It just delegates to an instance of the entity data model ObjectContext that is created as a member by the base class and uses it to return all the Tasks. If you want to modify that query to order, filter, or page the results, as long as you return an IQueryable<T>, WCF RIA Services will be able to expose that as a service and generate the client code to consume it.

Rebuild the solution to make sure all is well and to generate the client side code. If you Show All Files in the TaskManager project, you will see a Generated\_Code folder with a TaskManager.Web.g.cs file in it. This contains the generated code. I’ll dig into what is there in more detail in the next chapter.

### Step 4: Retrieve Data into the UI with DomainDataSource

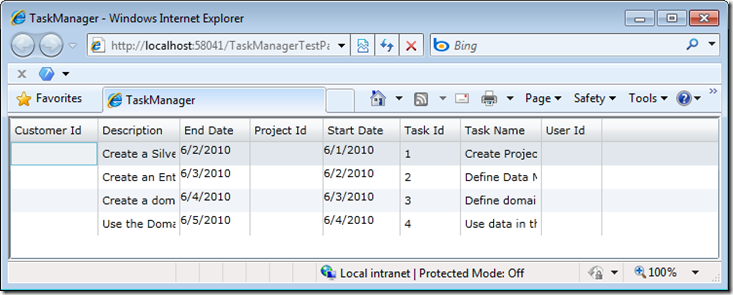
For this step, you will leverage the drag and drop capabilities of WCF RIA Services using the DomainDataSource directly in the UI. In a later chapter I’ll discuss the MVVM pattern, why this is not the best architecture for complex applications, and how you can still leverage the DomainDataSource but do so behind the scenes in a local service.

For this chapter though, open MainPage.xaml in the TaskManager project. The designer will open. Open the Data Sources window (Data menu > Show Data Sources). It will take a moment the first time you open it after the client code generation, so be patient. But you should see entities for the types returned by the domain service listed in the Data Sources window.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/DataSources_2.png)

Drag and drop the Task entity type onto the MainPage.xaml in the designer. A DataGrid will be generated with appropriate columns for each of the properties on the entity type, and a DomainDataSource will also be declared and hooked up to the GetTasksQuery method on the TasksDomainContext class. You can drop down in the XAML and remove all the sizing and positioning properties from the DataGrid (Height, Width, HorizontalAlignment,VerticalAlignment, and Margin) so that the DataGrid fills the containing Grid.

Build and run the application. You should see something like the following.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/RunningApp_2.png)

### Summary

In this chapter, I covered the basic concepts behind a WCF RIA Services application and stepped you through the simplest possible WCF RIA Services application. You created an Entity Data Model, a RIA Services DomainService, and consumed that service in the client using the DomainDataSource and the generated TasksDomainContext. In the rest of the chapters in this ebook, I’ll drill into more details on each aspect of WCF RIA Services, including real world application concerns of UI patterns, security, concurrency, testability, debugging and other aspects.

[Sample Code](file:///C:\Users\rgeorgieva\AppData\Roaming\Microsoft\Word\source_code\TaskManagerPart1.zip)

# Chapter 2: Querying Data

### Overview

In chapter 1, I introduced the basics of WCF RIA Services and walked you through the “hello world” equivalent for WCF RIA Services. That may have all seemed too simple, and you can’t really drag and drop your way to real world applications. For real applications, you are going to need to know more how to customize what services are exposed to the client and how to consume them on the client side. In this chapter, I’m going to drill down a little deeper into the process of querying data from the client. I’ll cover the conventions for query methods that you define on your domain service, as well as how to do it with configuration (attributes) instead of based on convention. I’ll show some of the programmatic ways that you can perform the queries on the client side. Finally, I’ll talk about using data from sources other than Entity Framework.

The starting point for this chapter is the solution that resulted from the steps in Chapter 1.

[Sample code](source_code/TaskManagerPart2.zip)

### Step 1: Add a parameterized collection query to your domain service

In the last chapter, I walked you through creating a domain service and after you completed the wizard, a single query method had been added to the domain service class that looked like this:

public IQueryable<Task> GetTasks()

{

return this.ObjectContext.Tasks;

}

That query method just returned all of the tasks in the database as an **IQueryable<T>** and took no arguments. If you want to have more specific queries available to the client without returning all the rows, you can add specific query methods to the domain service, and they will be code generated on the client side as well. You can also choose to return an **IEnumerable<T>**, but if the underlying provider (i.e. Entity Framework or LINQ to SQL) returns **IQueryable<T>**, they you should pass that through instead as it supports more scenarios.

Say that a common use case for your application was to allow the user to retrieve tasks for a given date range. They could search for tasks before a given start date or after or between. To support that, add the following query method to your domain service:

public IQueryable<Task> GetTasksByStartDate(

DateTime lowerDateTimeInclusive,

DateTime upperDateTimeInclusive)

{

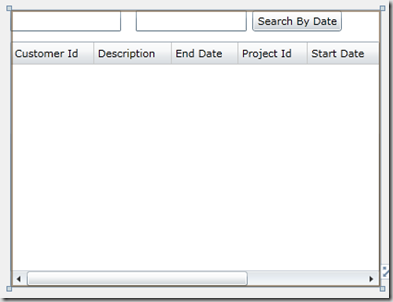
return this.ObjectContext.Tasks.Where(

t => t.StartDate >= lowerDateTimeInclusive && t.StartDate <= upperDateTimeInclusive);

}

### Step 2: Add UI to execute the query

To call the search query instead of the GetTasks query that the **DomainDataSource** in the UI currently calls at page load, you need to add some UI. Open up MainPage.xaml and click on the **DataGrid** that is filling the UI. Grab the top edge and drag it down a bit to make room for a few controls. Drag and drop two **TextBoxes** and one **Button** so that it looks something like the screen shot below.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/UIModification_2.png)

Name the first TextBox lowerDate and the second upperDate and the button searchButton. Set the Content of the button to Search By Date. Double click on the button to add a handler.

### Step 3: Execute a query on the client through the DomainContext

If you remember from the introduction in Part 1, the **DomainContext** is the code generated client side counterpart to the server side domain service. In this case, since the domain service is called **TasksDomainService**, the client counterpart is called **TasksDomainContext**. It exposes the ability to call the server side asynchronously and to also submit changes made to entities on the client side. The **DomainContext** is really the brains of the WCF RIA Services on the client side and has a lot of stuff going on internally. Whenever you retrieve entities from the server by executing a query through the domain context, the domain context also holds a reference to those objects along with some change tracking information. So if you modify those objects, it knows which have changed and can send just those entities back to the server side when you decide to submit those changes. But I’ll get into that some more in the next chapter. For now, let’s stay focused on the retrieval side of things.

In your search button click handler, you could add the following code:

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

{

DateTime lowerDateVal;

DateTime upperDateVal;

GetDates(out lowerDateVal, out upperDateVal);

TasksDomainContext context = new TasksDomainContext();

taskDataGrid.ItemsSource = context.Tasks;

EntityQuery<Task> query = context.GetTasksByStartDateQuery(lowerDateVal, upperDateVal);

LoadOperation<Task> loadOp = context.Load(query);

}

The important code here is the last four lines. To call the server side, you need an instance of a domain context. In a real application, you are going to want to create an instance of the domain context that you keep around for a while, because that is where the change tracking information for submitting changes resides. So you will typically move it to a member variable in a view model or possibly an application scoped service. But I’ll get into more detail on that in Chapter 4 of the ebook.

The domain context exposes collections of entities that include the sets of collections returned by query methods on your domain service. So far the domain service just exposes a collection of Tasks. Notice that the code is replacing the ItemsSource of the **DataGrid** with the Tasks collection on this domain context before making any calls against it. This is because when you query through the domain context, the queries execute asynchronously and will replace the contents of the exposed collection when the query returns from the server. That collection implements **INotifyCollectionChanged**and will raise events that causes the **DataGrid** (or any data bound control) to refresh itself when those events fire.

The code then gets an **EntityQuery<Task>** from the context with the arguments that are passed to the corresponding domain service method. That just sets up what you want done, it doesn’t actually make a call. Finally, it gets a **LoadOperation<Task>** from the context by calling Load. This is the point where a call is actually made to the server, on a background thread, and automatically marshaling the changes onto the UI thread to modify the collection when the results come back.

This is the same thing that is happening under the covers on the **DomainDataSource** that was used in the XAML in Chapter 1.

The GetDates method above just extracts the dates from the TextBoxes, checking for empty strings.

### Step 4: Add query methods that return a single entity

To return a single entity instead of a collection, you just define a method that does so on your domain service. A corresponding query method will show up on your domain context after compiling.

public Task GetTask(int taskId)

{

return this.ObjectContext.Tasks.FirstOrDefault(t => t.TaskId == taskId);

}

### Convention Over Configuration

This is an emerging trend in .NET development– reduce the amount of explicit configuration you need in your code and rely on some naming conventions. It may not be apparent from the domain service code so far, but it has actually been relying on convention. It is not obvious on the query side of things, because any method name will work. But for update, insert, and delete methods, there are a set of named prefixes WCF RIA Services is looking for, such as UpdateTask, InsertTask, and DeleteTask (or a number of variants for each operation).

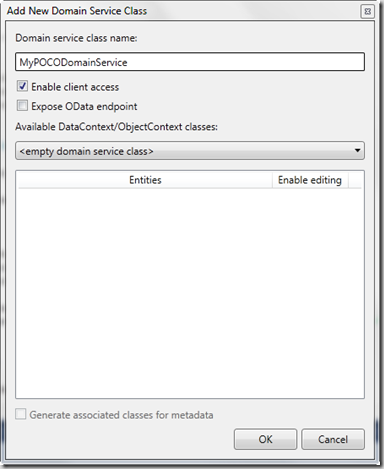
You can also use “configuration” by decorating the methods with attributes to indicate what kind of operation it is. For the Query side, the attribute is named, not surprisingly, **[Query]**. Using the attribute can make your code more explicit by making it easy to recognize at a glance what kind of method it is. But the**IQueryable<T>**, **IEnumberable<T>** or Entity return type makes it pretty obvious it is a query method anyway. The one advantage of using the Query attribute is that it supports a couple of additional things, such as specifying a maximum number of results (entity count) to return even if the underlying query returns more from the database.

### Custom Domain Services

What if you don’t want to use Entity Framework? Another choice is LINQ to SQL. Personally, I would recommend that if you are starting a new project, you focus on Entity Framework. It is really the most capable and likely to be long-lived data access approach for .NET now and in the future. LINQ to SQL is also supported through the separate WCF RIA Services Toolkit if you really want to use that instead.

But a lot of people are using other data access strategies (i.e. nHibernate) and other data sources (Oracle, MySQL, etc.). You can easily use those with WCF RIA Services as well, as long as you can define the data objects you want to pass as simple entities and populate those using whatever code you need to. These are referred to as POCO (Plain Old CLR Objects) domain services.

To do this, you just derive your domain service class from **DomainService** directly, instead of using the **LinqToEntitiesDomainService** base class. In the wizard when you Add > New Item > Domain Data Service, just select “empty domain service class” in the Available DataContext/ObjectContext classes drop down. Then define your own entities, return **IEnumerable<T>** collections of those entities for queries if your underlying data source does not support **IQueryable<T>**, and do whatever you need to do based on your data source in the methods.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/POCODomainService_2.png)

Then you would define your service in terms of your underlying data source and the entities that you are working with. The entity types you work with need to have properties that are other entities or a set of supported types that include all the built in types of the .NET framework.

The key (pun intended) thing that distinguishes your entities is that they need to have a key properties that uniquely identifies them. This is typically an int or Guid. You indicate that property with the **[Key]** attribute.

The code below shows a simple POCO domain service and its entity type.

public class Foo

{

[Key]

public int FooId { get; set; }

public string Name { get; set; }

}

[EnableClientAccess()]

public class MyPOCODomainService : DomainService

{

public IEnumerable<Foo> GetFoos()

{

return new List<Foo> { new Foo { FooId = 42, Name = "Fred" } };

}

}

In addition to the **[Key]** attribute, if you have properties on your entity that are entity types themselves (related entities or associations), then you will need to have a property alongside that entity that is an ID for that entity that gets set to its key field. Additionally, on the entity property itself, you will need an **[Association]** attribute to indicate the ID property that sets up the relation and indicate that it is a foreign key. On the entity property, you will also need an**[Include]** attribute to have that related entity also retrieved when the parent entity is retrieved to the client by RIA Services. See the MSDN Library documentation for these attributes for more information.

### Summary

In this chapter, you got a taste of how queries work with WCF RIA Services. You saw one way to query programmatically and bind to the results. You saw how to define additional query methods and learned about the conventions and configuration options for indicating your query methods in your domain service. You also learned about exposing POCO domain services. In the next chapter, I’ll show how to make changes to entities on the client side and get those changes made on the server.

[Sample code](source_code/TaskManagerPart2.zip)

# Chapter 3: Updating Data

### Overview

In Chapter 2, I covered the query capabilities of WCF RIA Services in a little more depth. In this chapter, I’ll focus on the updating side of things, which is a bit more complicated than just retrieving data because there is a lot more going on. I’ll be covering the basic coding patterns for updating data, as well as talking about what is going on behind the scenes. Before I get to that though, there is one more important aspect of querying that I need to cover that didn’t fit in the last chapter, or the WCF RIA Services black helicopters will come down and get me. :)

Get the starting point for this chapter [from Part 2 here](source_code/TaskManagerPart2.zip).

[Sample code](source_code/TaskManagerPart3.zip)

### IQueryable<T> and the Magic of Expression Trees

Back in Chapter 1, you defined a domain service method that looked like this:

public IQueryable<Task> GetTasks()

{

return this.ObjectContext.Tasks;

}

Now if you stare straight at this method, you would probably convince yourself that it is always going to return all of the Tasks in the database every time you execute it. However, if you tilt your head to the side just right, you’ll be able to see what it really does. Go ahead, I’ll wait… See it?

OK just kidding. To understand why this does not always return all tasks, you need to understand a little about expression trees and deferred execution. When GetTasks gets called, it is not actually executing a query at all. It is just forming an expression tree and returning that as an **IQueryable<T>** that describes what could be returned from this method. In this case, it could potentially return all tasks in the Tasks table. The expression tree that describes that is what gets returned to the client. The execution of actually running the underlying query is deferred until some future point when you actually try to act on the collection that the expression tree represents. Whenever you see an **IQueryable<T>**, remind yourself that it is not necessarily a collection yet – it is an expression tree that, when evaluated, will return a collection. But that expression tree can be modified by the receiver after returning it, and that can change the results that populate the collection when it gets evaluated.

The expression tree that is sent to the client can be modified before actually executing it. For example, suppose I execute the following code on the client.

TasksDomainContext context = new TasksDomainContext();

taskDataGrid.ItemsSource = context.Tasks;

EntityQuery<Task> query = context.GetTasksQuery();

LoadOperation<Task> loadOp = context.Load(query.Where(t=>t.TaskId == 1));

In line 2, I am binding to the Tasks collection on the domain context, which is currently empty because I just constructed the domain context. Then I get an **EntityQuery<Task>** from the context. This still doesn’t execute anything on the server, but the **EntityQuery<T>** lets me shape the expression tree based on what the target server method can execute (all tasks). When I call Load on the domain context, notice that I am passing a modified expression tree that includes a Where filter to only return a single task based on its ID. If you make a modification to the query client side like this, that expression tree is sent to the server when the async load operation executes, and that expression tree is applied to the actual expression tree that executes to retrieve rows. That means that not only won’t all rows be returned from the server to the client (saving bandwidth), but they won’t even be returned from the database when the query runs through the Entity Framework **ObjectContext** on the server side. You can convince yourself of this by firing up SQL Profiler and look at the query that runs on the database server. It includes a where clause for the TaskId column, as specified on the client side, without needing to explicitly declare a separate method that can do the filtering itself on the server side. Is that cool or what!?

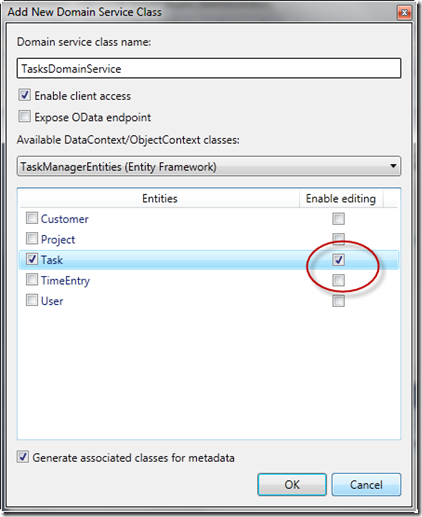
### DomainContext Caching and Change Tracking

OK, with that out of the way, let’s switch focus to updating data. Before I start walking through some code, it’s important to understand a few concepts about the way WCF RIA Services works. Behind the scenes of the domain context, there is a lot more going on that just proxying calls to the server. Any entities or collections of entities you retrieve are cached through the domain context on the client side. That is why in the code in the previous section I could just bind the data grid to the Tasks collection on the domain context before executing the query. That collection will fire change events and update the UI when the cached entities change, at the point where the async load completes. In addition to caching the entities, the domain context maintains change tracking information about the cached entities so it knows if any have been modified, deleted, or added.

So instead of explicitly calling the server every time you want to change an object, you can make a bunch of changes to a bunch of objects and those will all be cached and tracked by the domain context. Then when you want to get those changes persisted on the server, you call **SubmitChanges**.

### Step 1: Add update methods to your domain service

In Part 1 I walked you through the wizard for adding a domain service to the web project. At that point in the interest of keeping it as simple as possible, I didn’t have you do an important step for real apps – enable editing. When you add your domain services, you should check the box next to each entity that you expect to change at all on the client side:

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/_AddDomainService_2.png)

When you check this box, the following additional methods are added to the domain service, which you can also easily add by hand:

public void InsertTask(Task task)

{

if ((task.EntityState != EntityState.Detached))

{

this.ObjectContext.ObjectStateManager.ChangeObjectState(task, EntityState.Added);

}

else

{

this.ObjectContext.Tasks.AddObject(task);

}

}

public void UpdateTask(Task currentTask)

{

this.ObjectContext.Tasks.AttachAsModified(currentTask, this.ChangeSet.GetOriginal(currentTask));

}

public void DeleteTask(Task task)

{

if ((task.EntityState == EntityState.Detached))

{

this.ObjectContext.Tasks.Attach(task);

}

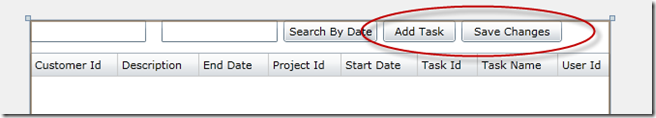
this.ObjectContext.Tasks.DeleteObject(task);

}

These methods are simple wrappers around the appropriate update operations on the entity framework object context.

### Step 2: Add some UI to invoke an add operation

Next you need something in the UI to invoke an update or add type of operation. Again, the pattern for update, add, and delete is pretty much the same – make the modification to the collection of objects exposed by the domain context, and call **SubmitChanges**. To keep it simple, you can just add another couple buttons to the top of the existing MainPage.xaml. Name the two buttons **addTaskButton** and **saveChangesButton**, set their Content property to Add Task and Save Changes respectively. Add button click handlers for both as well.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/AddTaskButton_4.png)

<Button Name="addTaskButton" Content="Add Task" Click="addTaskButton\_Click" .../>

<Button Name="saveChangesButton" Content="Save Changes" Click="saveChangesButton\_Click" .../>

### Step 3: Create a new Task, add it to the domain context, and submit changes

Add the following code to the event handlers, as well as adding a member variable to contain the domain context:

TasksDomainContext context = new TasksDomainContext();

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

{

taskDataGrid.ItemsSource = context.Tasks;

context.Load(context.GetTasksQuery());

Task newTask = new Task

{

TaskName = "Deploy app",

Description = "Deploy app to all servers in data center",

StartDate = DateTime.Today,

EndDate = DateTime.Today + TimeSpan.FromDays(7)

};

context.Tasks.Add(newTask);

}

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

{

context.SubmitChanges();

}

First, the code adds a member variable to the view for the domain context. As I mentioned earlier, when you are making changes to entities, you need to keep the context around long enough to submit the changes. In this case, I separated the add from the submit so you can see the effects on the client side. When you click the Add Task button, the **addTaskButton\_Click** handler first replaces the **ItemsSource** for the **DataGrid** to replace the **DomainDataSource** that was hooked up in Part 1 (this is just to keep the code isolated for this chapter). It then calls Load again to get this domain context populated with the entities from the back end. Then it creates and populates a new Task entity and adds it to the context’s Tasks collection. That change will be seen immediately by the UI because the underlying collection implements **INotifyCollectionChanged**. However, the change is just cached client side, so until you call SubmitChanges, the new entity is not sent to the back end. I had you separate that out to a separate method so you can see this in the UI.

When you click Add Task, you should see the new entity show up in the **DataGrid**, but its **TaskId** is initially set to zero. After you click the Save Changes button and **SubmitChanges** is called, you will see the **TaskId** update. That is because when **SubmitChanges** is called, the domain context makes an asynchronous call to the server, sending only those entities that have actually changed (modified, added, or deleted). After that, if the entity is modified server side, such as setting the **TaskId** at the DB level, the updated entity is returned to the client and is merged with the cached copy, making it so the one in the Tasks collection is current with what the back end has. Because the entity has changed, and it implements the **INotifyPropertyChanged** interface, the UI updates.

### DomainContext’s Async API

As mentioned in the previous parts, the Load and **SubmitChanges** API of the domain context execute asynchronously, meaning they don’t block the calling thread. They grab a thread from the thread pool behind the scenes, go make the calls to the server in the background, and when those calls complete, the work is automatically marshaled back to the UI thread to modify the entity collections and update the UI.

That is all smooth and effortless if all goes well. But the real world is not so perfect. You could have a connection burp, some knucklehead could have messed up the connection string, or you could have concurrency conflicts on the back end. Additionally, you may need to know when the operation completes so that you can do whatever is next in your workflow. To address these scenarios, you have two options: use the return type or hook up a callback that will be called when the operation is complete.

The first option is to work with the return value from these async methods. **Load** returns a **LoadOperation**, and **SubmitChanges** returns a **SubmitOperation**. These both derive from a common base class **OperationBase**, and they offer a bunch of information about the operation that you can write code against while it is running or after it is completed. They publish a **Completed** event when the operation completes, so you can subscribe to that to be notified when it is done. They expose any errors that happened and various other flags and properties you can check for results.

As an alternative to subscribing to the **Completed** event on the return value, you can instead call an overload of the Load or **SubmitChanges** operations that takes an **Action<LoadOperation>** or **Action<SubmitOperation>**, respectively. You pass in a callback reference and the method will be called when the operation is complete. Otherwise, the argument is the same type as the return type, so the same information is available to your code at that point.

### Summary

That’s all there is to it. Make changes to the entity collection on the domain context, and call **SubmitChanges**. Implement methods in the domain service that take those changes and push them to storage. There is a bit more to talk about with transactions and concurrency conflicts, but I’ll have to cover those in a future episode for space considerations. In the next chapter I’ll talk about how all this fits into the popular MVVM pattern.

[Sample code](source_code/TaskManagerPart3.zip)

# Chapter 4: Integrating with the Model-View-ViewModel Pattern

### 

### Introduction

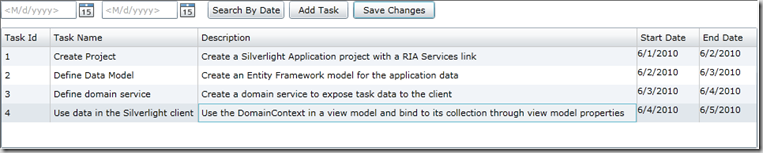
The Model-View-ViewModel pattern (MVVM) is a very popular approach for building more loosely coupled Silverlight and WPF applications. I won’t attempt to full define and explain the pattern here, but will give a quick explanation of what it is about and why you would want to use it. At a high level, MVVM is an alternative or evolution of MVC and MVP patterns, but taking full advantage of the rich data binding, commands, and behaviors that we have available to us in Silverlight and WPF. The view model’s primary responsibility is to offer up state to the view in the way the view wants to see it. It exposes properties that the view can easily bind to that let the view display the data it needs and to also hook up commands and behaviors that point back to the view model to invoke interaction logic that the view model encapsulates. The view model is all about containing and manipulating the data that the view needs and providing the interaction logic to support the view. The view model acts as a translator from the complex world of the overall application model to the specific set of data that a single view needs, structured in the way the view needs to display it. Structurally, you typically set the view’s DataContext equal to an instance of the view model so that the view can easily bind to the exposed properties of the view model. There are many ways to get this hook up done.

The motivations for using it include the fact that it allows you to work on the view and the view model mostly in isolation, facilitating developer/designer workflow. Additionally, by separating your interaction logic into the view model, you can more easily unit test that logic because it is not tightly woven with the UI itself. And finally, the pattern just provides a clean separation of responsibility so the view can just be about the structure and visual behavior of what you see on the screen, and the view model can just be about the data and logic surrounding the data that the view needs to support it. For more in depth coverage, I recommend you read Josh Smith’s excellent [article on MVVM](http://msdn.microsoft.com/en-us/magazine/dd419663.aspx), as well as the guidance being developed by the Microsoft patterns & practices Prism team in Prism 4 at <http://prism.codeplex.com/>. Prism 4 is due out in the September/October timeframe but there are already public drops in the downloads section on CodePlex.

So now let’s see how MVVM fits with WCF RIA Services. The starting point code for this chapter will be the sample code from Chapter 3 of the ebook, which [you can get here](source_code/TaskManagerPart3.zip). You can get the [finished code for this chapter here](source_code/TaskManagerPart4.zip).

### Step 1: Factor out a view model

In the last chapter’s version of the TaskManager application, I was already starting to have a fair amount of code building up in the code behind of the view (the **MainPage** user control), even with such a simple application. The first step is to factor that code out into a view model. Create a new class named **TasksViewModel** in the project. Often a good way to get started defining your view model is to analyze the view and determine what properties would be needed to support it. The image below shows the form as it is currently defined. To clean things up a bit, I’ve replaced the two **TextBoxes** for date input at the top with **DatePickers**, removed the unused properties on the Task from the **DataGrid**, and reordered the columns to look a little better.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/figure1_2.png)

By looking at the UI design, you can identify 6 properties the view model should expose to support this view: two **DateTime** properties for lower and upper dates for a search; three commands to support searching by date, adding a task, and saving changes; and one Tasks collection. In its simplest form, that would look like this:

public class TasksViewModel

{

public DateTime LowerSearchDate { get; set; }

public DateTime UpperSearchDate { get; set; }

public ICommand SearchByDateCommand { get; set; }

public ICommand AddTaskCommand { get; set; }

public ICommand SaveChangesCommand { get; set; }

public IEnumerable<Task> Tasks { get; set; }

}

However, you generally need to implement **INotifyPropertyChanged** on your view model which requires the expanded syntax for each property so that the UI can stay fresh through its data bindings if the bound property on the view model changes, so each property definition should look more like this in its full form, except possibly the private set properties that the view model sets once and does not change.

public class TasksViewModel : INotifyPropertyChanged

{

public event PropertyChangedEventHandler PropertyChanged = delegate { };

DateTime \_LowerSearchDate;

public DateTime LowerSearchDate

{

get

{

return \_LowerSearchDate;

}

set

{

if (value != \_LowerSearchDate)

{

\_LowerSearchDate = value;

PropertyChanged(this, new PropertyChangedEventArgs("LowerSearchDate"));

}

}

}

To support the **ICommand** properties, you will need a decent command implementation. I’ve included a simple **RelayCommand**implementation, but normally in real projects I use the **DelegateCommand** from Prism.

A view model is all about managing the data needed by a view, and a **DomainContext** in RIA Services is all about providing and tracking changes to the data, so a simple approach to using RIA Services with MVVM is to simply use a **DomainContext** within your view model.

TasksDomainContext \_Context = new TasksDomainContext();

public TasksViewModel()

{

SearchByDateCommand = new RelayCommand<object>(OnSearchByDate);

AddTaskCommand = new RelayCommand<object>(OnAddTask);

SaveChangesCommand = new RelayCommand<object>(OnSaveChanges);

Tasks = \_Context.Tasks;

if (!DesignerProperties.IsInDesignTool)

{

\_Context.Load(\_Context.GetTasksQuery());

}

}

Here I have made the **TasksDomainContext** a member of the view model, and initialized the commands in the constructor to point to methods within the view model. The Tasks property exposed by the view model just holds a reference to the Tasks entity collection exposed by the domain context, which will raise **INotifyCollectionChanged** events to keep the view fresh when the collection changes, which will happen on the initial load, when you add Tasks, and when **SubmitChanges** completes and has updated entities from the back end. Notice the use of the **DesignerProperties.IsInDesignTool** property to prevent calling Load in the designer, which would break it.

Next is to move the methods that were in the code behind of **MainPage** into the view model. I’ve cleaned them up a bit in the process too. Note that the search method now leverages the deferred execution I talked about in Part 3, making the **GetTasksByStartDate** domain service method unnecessary since the client can specify that search expression itself. I’ve also moved the creation of a new Task into a simple popup **ChildWindow** so you can actually edit the values. Note the commend in the code below – showing a dialog from a view model is really a no-no, just doing it here because the focus is on RIA Services, not on full MVVM patterns.

private void OnSearchByDate(object param)

{

\_Context.Tasks.Clear();

EntityQuery<Task> query = \_Context.GetTasksQuery();

LoadOperation<Task> loadOp = \_Context.Load(query.Where(t => t.StartDate >= LowerSearchDate && t.StartDate <= UpperSearchDate));

}

private void OnAddTask(object param)

{

// Generally don't want to do this for testability reasons

// Simplification because MVVM structuring is not the focus here

// See Prism 4 MVVM RI for a cleaner way to do this

AddTaskView popup = new AddTaskView();

popup.DataContext = new Task();

popup.Closed += delegate

{

if (popup.DialogResult == true)

{

Task newTask = popup.DataContext as Task;

if (newTask != null) \_Context.Tasks.Add(newTask);

}

};

popup.Show();

}

private void OnSaveChanges(object param)

{

\_Context.SubmitChanges();

}

At this point you have you have a functioning view model, now I’ll clean up the view to work with the view model.

### Step 2: Hook up the view to the view model

From Part 1, the view was using the **DomainDataSource** that was generated from the drag and drop. If you want a clean MVVM design, you will unfortunately have to pass on using **DomainDataSource**. That object is effectively putting state management and query logic into the XAML itself, which violates the clean separation of responsibilities followed in MVVM.

So basically I took the existing view, deleted out the **DomainDataSource**, and added appropriate bindings to each of the controls to bind to the exposed view model properties as shown below. For a top level view like this, constructing the view model in the XAML to set it as the data context is a simple way to get the view’s data context set to an instance of the view.

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

<UserControl.DataContext>

<local:TasksViewModel/>

</UserControl.DataContext>

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

<sdk:DataGrid ItemsSource="{Binding Tasks}" .../>

<Button Command="{Binding SearchByDateCommand}" .../>

<Button Command="{Binding AddTaskCommand}" ... />

<Button Command="{Binding SaveChangesCommand}" ... />

<sdk:DatePicker SelectedDate="{Binding LowerSearchDate}" ... />

<sdk:DatePicker SelectedDate="{Binding UpperSearchDate}" ... />

</Grid>

</UserControl>

Lastly, remove all the code from the code behind of the view, since it is no longer being used (all the button click handlers were removed in the process of hooking of the commands with bindings to the view model properties).

public partial class MainPage : UserControl

{

public MainPage()

{

InitializeComponent();

}

}

### Summary

The simplest way to use WCF RIA Services with the MVVM pattern is to stop using the **DomainDataSource** in your views and to use a domain context as the repository of data for your view models. You can still leverage the drag and drop Data Sources window features; you just have to do a little clean up to delete off the **DomainDataSource** after the drop and touch up the bindings a little.

There are two downsides to the approach as I have shown it here: testability and separation of concerns. One of the benefits of the MVVM pattern is supposed to be unit testability. But with a concrete type dependency on the domain context type in your view model, you are out of luck for mocking out that dependency. There are two approaches possible to address this:

1. Use dependency injection to provide the domain context to the view model, and create a mock **DomainClient** and pass it to the **DomainContext**. This allows your view model to keep using the full functionality of the **DomainContext**, but you are able to mock out the calls to the service underneath the **DomainContext**. This approach is outlined [in this post](http://www.nikhilk.net/NET-RIA-Services-ViewModel-Pattern-2.aspx). I’ll be providing an example in chapter 8 of this ebook, which is focused on testability.
2. Factor out the **DomainContext** to a repository service that defines an interface similar to the **DomainContext** API and consume that from the view model. This would allow you to mock your repository service that contains the domain context. It would also address the separation of concerns problem, because then you would be separating the implementation choice of using RIA Services from the view model, which should really be separated. However, you would give up a lot of flexibility in the view model and it would not be an easy repository implementation.

Neither of these approaches is particularly easy. I’ll cover them in more detail later in chapter 8, and may blog some more on this topic as well. That’s all for now.

[Sample code](source_code/TaskManagerPart4.zip)

# Chapter 5: Metadata and Shared Classes

### 

### Introduction

In this chapter, I am going to quickly cover the metadata and shared code facilities of WCF RIA Services. In a nutshell, metadata classes allow you to add functionality in the form of attributes to the code generated entities that you are using in your services and on the client. You can add attributes that causes validation to happen, which I will go into more detail on in the next chapter, as well as attributes that affect the way entities are related and transmitted across the wire by RIA Services. With shared code, you can define chunks of logic or partial class extensions for your entities that are defined once on the server side, but get code generated on the client side as well so that the same logic can be used in both places.

For this chapter and moving forward with the rest of the ebook, I am going to depart from the step-by-step instruction format for the chapters. Because I am building and adding functionality to the application with each chapter to show some of the more sophisticated capabilities of RIA Services, I cannot easily lead you through writing all the code without writing an unnecessarily long tome for each chapter. So I’m going to switch to a more descriptive format where I will simply highlight the appropriate sections of the code for the concepts discussed in the sections of the chapter.

You can find the [completed code for this chapter here](source_code/TaskManagerPart5.zip). In this chapter’s version of the application, I have added the ability to display and add time entries associated with a task.

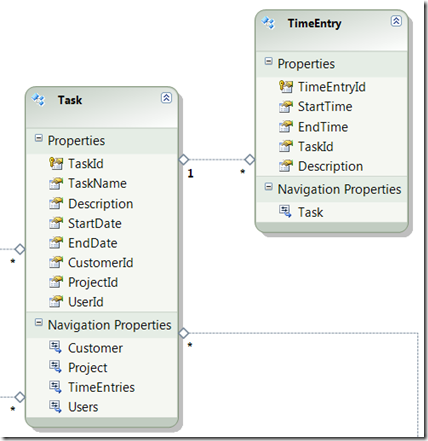
### Metadata Classes

When working with WCF RIA Services, your client side entities are code generated at compile time as discussed in earlier chapters in the ebook. At a simplistic level, WCF RIA Services tools reflect on the compiled entity types at compile time and generate entities with the same set of properties on the client side. However, any methods or behavior embedded in the property definitions that are part of the entity type on the server side are not carried over to the client side. If you are working with POCO entities, many attributes on your properties will also be regenerated on the client side. Those attributes can include validation attributes from the data annotations namespace, or they can include certain attributes that are used by RIA Services to influence how it tracks changes on related objects and which related objects it transfers across the wire.

The challenge is that if you are using a technology like Entity Framework, the entity classes themselves are generated code that you should not be directly modifying because it will be regenerated if you make modifications to the model in the designer. So you can’t really add attributes to the entity definitions themselves. You might start to think - “well, I can just use partial classes to add what I want”. But you would be wrong. While it is easy to add methods and new properties to a class via partial classes, you cannot really modify existing properties via a partial class extension.

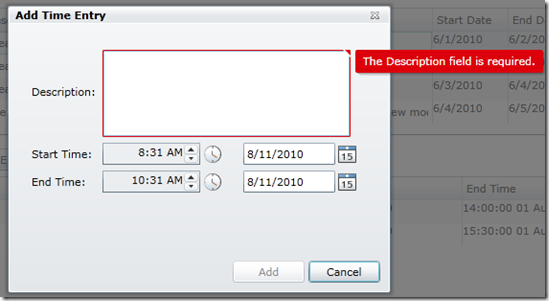
Along come metadata classes to the rescue. A metadata class, or “buddy class” as it is called, is a way to solve this conundrum. A metadata class is a simple class that defines the same or a subset of the same properties defined by one of your entity classes, but those properties have no implementation and won’t actually be called at all. They are just a place where you can add attributes, and those attributes will be added to the corresponding properties in the code generated client entities. Depending on what the attributes are, they will also be used by RIA Services on the server side.

Let me show an example. In the sample application for this chapter, I have added the ability to display and add child **TimeEntry** objects to a Task object. In the Entity Data Model for the sample, the Task type has a 0 to many relationship with a set of child **TimeEntry** types. In the object model, this looks like this:

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/8-11-2010%208-28-57%20AM_2.png)

As mentioned, these entities are defined through code generation by Entity Framework, so I don’t have the opportunity to modify them.

If I want to make the Description property on a **TimeEntry** a required field, I can use the **[Required]** attribute from the System.ComponentModel.DataAnnotations namespace. If I do that, I would get automatic validation of that in the UI so I end up with a UI that looks like this:

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/8-11-2010%208-31-38%20AM_2.png)

I can actually get this pretty much “for free” with RIA Services and Silverlight by just adding a **[Required]** attribute to the **Description** property on the **TimeEntry** type. However, I can’t put it in the entity code generated by Entity Framework, so I need a metadata “buddy” class to add it to the entity.

My metadata class looks like this:

[MetadataTypeAttribute(typeof(TimeEntry.TimeEntryMetadata))]

public partial class TimeEntry

{

internal sealed class TimeEntryMetadata

{

// Metadata classes are not meant to be instantiated.

private TimeEntryMetadata()

{

}

[Required]

public string Description { get; set; }

}

}

The convention is to define a nested class inside a partial class extension for your entity type on the server side. You link the metadata class to the entity type through the **MetadataType** attribute. The metadata class can redefine any of the properties exposed by the entity type and add attributes to them. You do not have to redefine all of them, only those you want to add an attribute to. So in this case I just add the **[Required]** attribute to the **Description** property. These metadata classes can be generated for you when you run the initial Add Domain Service Class template. The “Generate associated classes for metadata” checkbox at the bottom of the Add Domain Service dialog will create a metadata class for each entity managed by your domain service and will redefine all of the properties in those entities within the metadata class so that you can easily just drop in an add attributes as appropriate.

The generated client entities will then contain these attributes:

public sealed partial class TimeEntry : Entity

{

...

[Required()]

public string Description

{ }

Other attributes that are important in the metadata class have to do with entity relations. When you define retrieval methods in your domain service for an entity type such as a Task, by default WCF RIA Services will not send related entities to the client. So even though the **Task** type has a property for a collection of child **TimeEntries**, even if my domain service loads all the child entities (with the Include() method in Entity Framework), those child entities will not be sent to the client. To get them sent, I first have to ensure that they get loaded from the data layer:

public IQueryable<Task> GetTasks()

{

return this.ObjectContext.Tasks.Include("TimeEntries").OrderBy(t=> t.StartDate);

}

And second I need to add an [Include] attribute on the corresponding property in the entity metadata class:

[MetadataTypeAttribute(typeof(Task.TaskMetadata))]

public partial class Task : INotifyPropertyChanged

{

internal sealed class TaskMetadata

{

// Metadata classes are not meant to be instantiated.

private TaskMetadata()

{

}

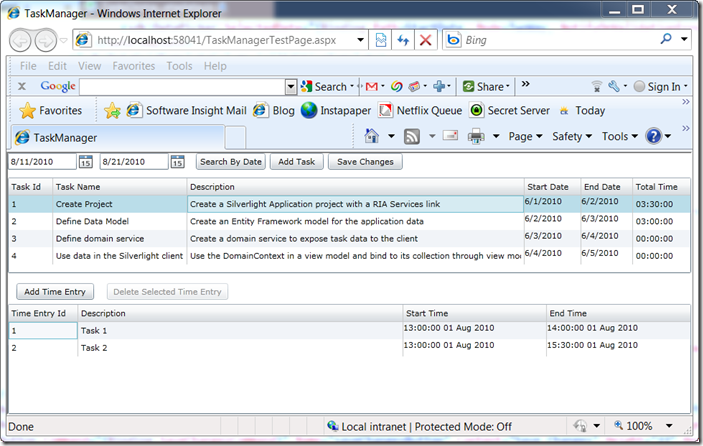
[Include]

public EntityCollection<TimeEntry> TimeEntries { get; set; }

}

}

Now the time entries get sent to the client side and I can display them (after adding some of course):

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/8-11-2010%208-21-15%20AM_2.png)

If you want to manage child entities as wholly owned child objects of the parent, you can also use the **[Composition]** attribute. Putting that on a property makes it so a change to a child object causes the parent object to be marked as modified. Then when you **SubmitChanges** on the client side both the parent and the changed child entities are sent along. You do have to do a little more work in your parent entity Update method because you have to handle the inserts, updates, and deletes of your children as well as those for the parent entity type.

### Shared Code

Another facility that comes in handy is the ability to define a chunk of code on the server side that can be used there, but that will also be available (through code generation) on the client side as well. All you need to do to leverage this is to put the code in a file with a \*.shared.cs (or \*.shared.vb) naming convention in the server project. The sample code for this chapter includes a file named Task.shared.cs containing a partial class extension for the Task entity type that adds a computed property:

public partial class Task

{

public TimeSpan TotalTime

{

get

{

TimeSpan total = new TimeSpan();

if (TimeEntries != null)

{

TimeEntries.ToList().ForEach(delegate(TimeEntry te)

{

total += (te.EndTime - te.StartTime);

});

}

return total;

}

}

}

This property could be used to support the business logic on the server side, and is used for display on the client side as you can see from the screenshot shown earlier.

Another common use of shared code is for custom validation rules that get tied in to a property through a **[CustomValidation]** attribute. I’ll show an example of that next chapter when I go into a little more depth on validation.

### Summary

Metadata allows you to add functionality to your server and client side entities by decorating properties in a metadata class with attributes that you want to affect the behavior of your entities. A metadata class is a class tied in to the entity type through the **[MetadataType]** attribute.

You can tie in validation attributes, as well as attributes such as the **[Include]** and **[Composition]** attributes that influence the way WCF RIA Services handles child entities. Shared code allows you to share code files between the client and server side without needing to copy and maintain two versions. All you need to do for that is name your code files with a .shared.cs (or .vb) naming convention.

Next chapter I’ll dig in to validation a bit more.

[Sample code](source_code/TaskManagerPart5.zip)

# Chapter 6: Validating Data

### Introduction

In Chapter 5 I discussed metadata classes and shared code, and one of the example uses of metadata classes I showed was using a **[Required]** validation attribute to cause some validation on both the client and server side. In this chapter, I will dive into more depth on the various ways you can validate data using WCF RIA Services. I’m going to look at four ways of validating input data:

* Using built-in validation attributes from the System.ComponentModel.DataAnnotations namespace
* Using custom validation attributes to invoke custom logic on the server and client side for cross property validation
* Using server side validation when changes are submitted to the back end
* Doing asynchronous validation using a domain service invoke operation

To demonstrate these things in the application context that I have been building up over the last 5 chapters, I needed to add some more functionality. That functionality included some small schema changes to the database, so a new database creation script is included with the chapter code to create an appropriate database schema with some minimal sample data to be able to run the application. Because I am continuing to keep the focus on WCF RIA Services capabilities and not on general Silverlight display functionality, the display of some of the validation errors in the sample is necessarily crude to keep the changes to the app to a minimum and to keep the chapter length digestible.

The new functionality includes the ability to add customers (and there are a couple defined in the DB script so you don’t have to add any to get started), as well as select a customer associated with a given **Task**. There are validations associated with the definition of a **Customer**, as well as for creating time entries associated with a task that has a customer, described in more detail later in the chapter.

The finished code for this chapter [can be taken from here](source_code/TaskManagerPart6.zip).

### Data Annotation Validation Attributes

As mentioned last time, WCF RIA Services has rich built-in support for the validation attributes in the **System.ComponentModel.DataAnnotation** namespace. These attributes include the ability to specify simple forms of validation on entity properties by adding an attribute to an entity property. If your entity model is one where you cannot modify the properties directly (such as the code generated entities of Entity Framework), you can place the validation attributes on the properties of your metadata class. Those validation attributes will be used for server side validation by WCF RIA Services, and they will also be used client side because the client generated entities will have those attributes on their properties.

The available validation attributes include:

* Required
* StringLength (added by Entity Framework for string fields and therefore generated on the client)
* Regular Expressions
* Range (for numeric inputs)

There are a few others in the namespace that you might want to look into that purely affect display, but I won’t go into details of those in this chapter.

To apply them when using an entity framework model, you again just add them to the properties of your metadata class.

The generated client code will include those defined in your metadata class, as well as any defined on the model entities themselves, such as the**[StringLength]** attributes added by Entity Framework for string fields in the database.

As an example of using one of these attributes, a **Customer** has a **Phone** property that is a string, but needs to follow a proper format for a phone number. A regular expression is a great way of checking this kind of thing. So the metadata class for my **Customer** entity type includes the following attribute on the **Phone** property:

[MetadataTypeAttribute(typeof(Customer.CustomerMetadata))]

public partial class Customer

{

internal sealed class CustomerMetadata

{

...

[RegularExpression(@"^\D?(\d{3})\D?\D?(\d{3})\D?(\d{4})$", ErrorMessage="...")]

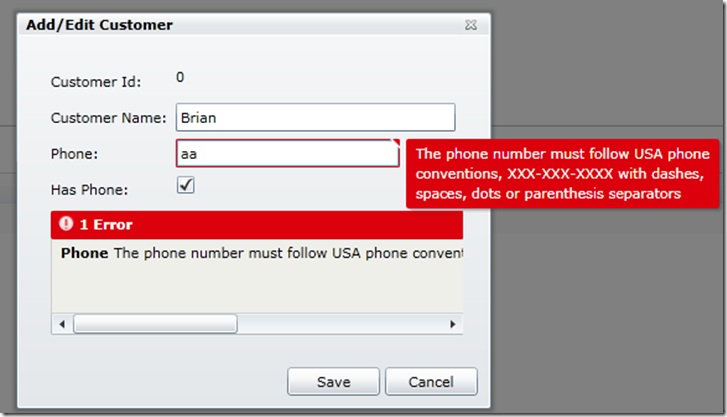
public string Phone { get; set; }

...

}

You can see the only challenge in using the regular expression validation is in coming up with the right regular expression. The one I am using here allows several formatting variants of US phone numbers (3 digits, 3 digits, 4 digits with variations on delimiters in between).

Just by having that attribute on the server entity or its metadata class, it shows up on the code generated client entity. Additionally, the WCF RIA Services **Entity** base class that is added to the client entity has an implementation of **INotifyDataErrorInfo** that uses the data annotation attributes to perform validation when data bound in the UI. So just by adding these attributes with appropriate error messages, you get validation indications for the user like shown below.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/8-24-2010%207-20-20%20AM_2.png)

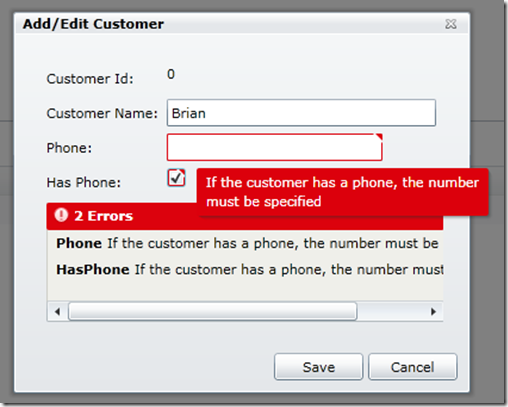
The error message shown on the **Phone** input **TextBox** happens automatically when the focus leaves the control. In the sample code, I used a behavior to cause the binding to update on each keystroke to get more immediate feedback to the user. The error message shown is the one specified in the server side attribute. You can also see that I added a **ValidationSummary** control to the form. That is to display when there is more than one validation error on the form at a time, which will come into play in the next section.

### Custom Validation Attributes

In addition to the pre-built validation attributes discussed in the last section, there is a**[CustomValidation]** attribute in the data annotations namespace that allows you to point to a static method you can define to perform any kind of custom logic you need to validate when a property is set. That may involve doing some calculation or conditional logic, but ultimately you are deciding whether the newly input values are valid. Keep in mind that if you put these attributes on the server side entity or its metadata class, you will need to ensure that the code defined in the custom validation will be able to execute on both the client and server side. So you would not want to put code in there that does a look up against the database for example. I’ll show examples of how to deal with that kind of validation in the next two sections.

One place custom validations come in very handy is for cross-property validations. When the validity of one property’s value depends on the value of another property on the same object, or possibly the values of some of the child objects or a parent object, you can accommodate this fairly easily using custom validation.

In the **Customer** class for the sample, I added a contrived property called **HasPhone**. The simple idea here is that if **HasPhone** is set to true, then a valid phone number should be supplied. If **HasPhone** is set to false, no phone number should be supplied. This validation should be triggered whenever either property is modified, and both properties can be displayed as having a validation error when they are out of sync. You can see this in action below. Notice that both input controls (Phone **TextBox** and HasPhone **CheckBox**) are indicating a validation error, and there are two errors shown in the **ValidationSummary** control.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/8-24-2010%207-23-02%20AM_2.png)

To support this, you first define the validation method that you want to point to from the **[CustomValidation]** attribute. That method has to have a particular signature, specifically it must be public, return a **ValidationResult**, and take in a value and a **ValidationContext**. If you are doing two properties with different types on the properties like I am here, you can break it down into two methods with each value having the appropriate type, even if they share some validation code:

public class CustomerPhoneValidator

{

public static ValidationResult CrossValidatePhoneWithHasPhone(string newPhone, ValidationContext context)

{

Customer customer = context.ObjectInstance as Customer;

if (customer == null) return ValidationResult.Success;

bool hasPhone = customer.HasPhone;

string phone = newPhone;

return ValidatePhoneAndHasPhone(hasPhone, phone);

}

public static ValidationResult CrossValidateHasPhoneWithPhone(bool newHasPhone, ValidationContext context)

{

Customer customer = context.ObjectInstance as Customer;

if (customer == null) return ValidationResult.Success;

bool hasPhone = newHasPhone;

string phone = customer.Phone;

return ValidatePhoneAndHasPhone(hasPhone, phone);

}

private static ValidationResult ValidatePhoneAndHasPhone(bool hasPhone, string phone)

{

if (hasPhone && string.IsNullOrWhiteSpace(phone))

{

return new ValidationResult(@"If the customer has a phone, the number must be specified",

new string[] { "HasPhone", "Phone" });

}

else if (!hasPhone && !string.IsNullOrWhiteSpace(phone))

{

return new ValidationResult(@"If the customer does not have a phone, the number should not be specified",

new string[] { "HasPhone", "Phone" });

}

else

{

return ValidationResult.Success;

}

}

}

You can see that I have two validation methods, one for the boolean **HasPhone** property and one for the string **Phone** property. Other than the type of the value, the signature of the methods is the same. You can see that the **ValidationContext** argument that is passed to your method allows you to get a reference to the whole object that the property being validated is on, allowing you to get to other properties on that object. At the point where your validation method is being called, the target property (i.e. **Phone**) has not been set on the object itself yet. The value being passed in to your validation method is the proposed new value, and it is up to your method to decide whether to allow it to be set or not. If you are fine with the value being set, you just return a **ValidationResult.Success** value. If you do not like the value, then you can return a **ValidationResult** with an error string populated and optionally a string array containing all of the property names affected by the error. This is how you can address cross-property validation scenarios like the one here. If no phone or an invalid phone number is supplied, I can check to see if the **HasPhone** property is set to true. If so, both **Phone** and **HasPhone** are in error because they have to agree with respect to their supplied values.

To apply the custom validation to a property, you just use a**[CustomValidation]** attribute:

[MetadataTypeAttribute(typeof(Customer.CustomerMetadata))]

public partial class Customer

{

internal sealed class CustomerMetadata

{

...

[CustomValidation(typeof(CustomerPhoneValidator),"CrossValidatePhoneWithHasPhone")]

[RegularExpression(@"^\D?(\d{3})\D?\D?(\d{3})\D?(\d{4})$", ErrorMessage="...")]

public string Phone { get; set; }

[CustomValidation(typeof(CustomerPhoneValidator), "CrossValidateHasPhoneWithPhone")]

public bool HasPhone { get; set; }

...

}

}

In this case I have a separate attribute for the regular expression and the cross-property validation for demonstration purposes. That will result in two different errors at times. For a more user friendly experience, as long as you are writing custom validation logic for a property tied in with the**[CustomValidation]** attribute, you might want to move that format validation inside the custom validation logic so that you only return one error at a time from a single control value.

### Server-Side Validation

Some kinds of validation are not appropriate to do on the client side. You may have data you need to look up on the back end that you don’t want to expose to the client that will drive the decision of whether a given chunk of data is valid or not. For the sample application, I added the ability to look up a customer status on a customer when a task is saved. If the customer status (which I don’t want exposed to the client so that it doesn’t get inadvertently displayed in the client) is marked as “DeadBeat” status, then I want to limit the number of billable hours I accrue for that customer. So at the point where I go to save a task on the back end, I need to look up the customer status and see how many hours have been added to the modified **Task** as child **TimeEntries** and decide whether to accept that task for update. If I don’t want to accept it, I can throw a **ValidationException** from the update method, and validation errors will be added to the entity in the client application and can be displayed in various ways.

To facilitate this scenario, it did require a change to my object model. I talked briefly about the**[Composition]** attribute in the last chapter. In order to have the **TimeEntries** sent down with the **Task** even if they have not changed themselves (such as when a customer has just been assigned to the **Task**), I needed to switch to having the **TimeEntries** collection on the **Task** be a**[Composition]** relationship. When you do that, the **TimeEntries** become wholly owned by their parent **Task**, and you don’t submit updates to individual time entries the way the application was doing in the last couple chapters. Now you just get updates to tasks, and those changes might involve changes to the discrete properties of the task, the assignment of a related customer (which is not a **Composition** relationship because customers can be independently queried and updated) or the addition/removal/change of a time entry within the child **TimeEntries** collection of the **Task**. You can check out the **UpdateTask** method in the domain service to see the way this altered the coding patterns. Basically you have to do some state detection yourself and call the appropriate parts of the data access API (Entity Framework in this case) to manage the inserts/updates/deletes of your child objects in a composition relationship.

By going with a composition relationship, though, now all child **TimeEntries** are sent down with a modified task and you can evaluate the package as a whole for server side validation. So the update and insert methods for **Tasks** call a helper method that first does the server side validation:

private void ValidateCustomerHours(Task task)

{

if (task.CustomerId.HasValue)

{

int customerId = task.CustomerId.Value;

bool timeAllowed = IsTotalTimeAllowed(customerId, task.TotalTime);

if (!timeAllowed)

throw new ValidationException(

"You cannot enter more than 10 hours per billing cycle for this customer.");

}

}

[Invoke]

public bool IsTotalTimeAllowed(int customerId, TimeSpan totalTime)

{

Customer relatedCustomer = ObjectContext.Customers.Where(c => c.CustomerId == customerId).FirstOrDefault();

if (relatedCustomer != null && relatedCustomer.Status == "Deadbeat"

&& (totalTime > TimeSpan.FromHours(10)))

{

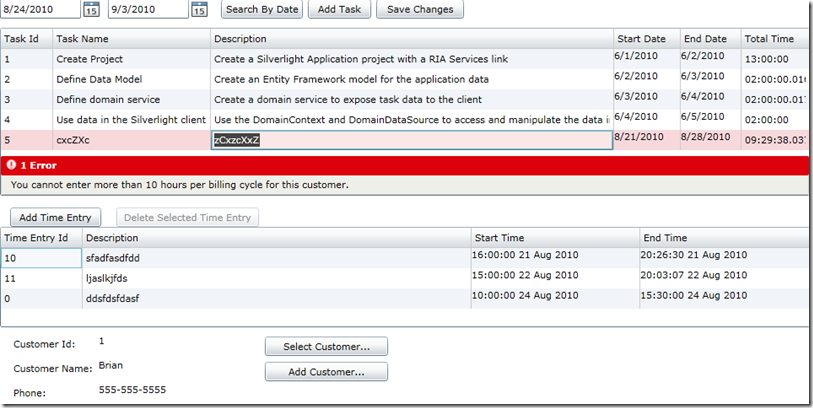
return false;

}

return true;

}

If I wrote the client so that it did not validate the max hours for a customer on the client side before sending the task to the back end for updating or inserting, this code would be invoked and would throw the **ValidationException** back to the client. On the client side, WCF RIA Services would add a **ValidationError** to the **Task** and it could be displayed by the build in validation mechanisms of Silverlight Controls.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/8-24-2010%2012-55-52%20PM_2.png)

### 

### Async Validation through Invoke Methods

When it comes to validation, it is generally better to let the user know as early as possible when they have entered invalid data instead of waiting until a later point such as when they have tried to send that invalid data to the back end for persistence. A decent way to do this in WCF RIA Services is to use an **[Invoke]** method on your service. In addition to the CRUD methods you can expose on a WCF RIA Service, you can also add arbitrary methods that you mark with an **[Invoke]** attribute. These too will be exposed as service methods to the client and will show up on the corresponding domain context. They can pass different types than the main entities that the domain service is managing as entities, but there is a restricted set of types that you can pass as parameters and return types. The WCF RIA Services enumerates the types that can be used as parameters and return types on domain service methods.

To support more immediate feedback to the user for the case of entering too many hours for a customer with bad status, I added the **IsTotalTimeAllowed** method that you can see in the code snippet in the previous section. That method is being used both internally by the server side validation, and can be called directly from the client as well. In the client side view model, at the point where the user has entered a new time entry, I can make a call to that service method to check whether the new time entry will exceed the max allowed hours.

TimeEntry entry = popup.DataContext as TimeEntry;

...

if (SelectedTask.CustomerId.HasValue)

{

TasksDomainContext context = new TasksDomainContext();

Action<InvokeOperation<bool>> action = delegate(InvokeOperation<bool> op)

{

...

if (entry != null) SelectedTask.TimeEntries.Add(entry);

...

};

context.IsTotalTimeAllowed(SelectedTask.CustomerId.Value,

SelectedTask.TotalTime + (entry.EndTime - entry.StartTime), action, null);

}

As you can see, the invoke method shows up on the domain context as a normal async service method. You can pass a callback delegate of type**InvokeOperation<T>**, where T is the return type of the invoke method, and that callback will be invoked when the async call is complete. You get the same thread marshaling back to the UI thread that you do with **Load** and **SubmitChanges** operations, so you don’t have to do any thread marshaling yourself.

You can see the full implementation with some (crude) user prompting in the sample application in the chapter source code.

### Summary

The built in validation support in WCF RIA Services is one of the more compelling features that makes me want to use WCF RIA Services for most data-oriented situations in Silverlight. The ease of using data annotation attributes to express simple validation rules or custom validation attributes to tie in more complex validations covers a large number of validation scenarios nicely. Add to that the ability to invoke whatever logic you need from your domain service CRUD methods and Invoke methods on the back end and you have a nice end to end solution for handling validation.

The fact that WCF RIA Services takes care of adding the validation errors to the entities on the client side and implements **INotifyDataErrorInfo** on those entities to participate nicely with the data binding features of Silverlight just rounds it all out into a very powerful part of WCF RIA Services.

[Source code](source_code/TaskManagerPart6.zip)

# Chapter 7: Authentication and Authorization

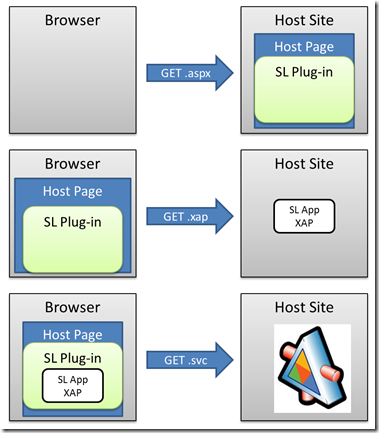
### 

### Introduction

In this chapter, I’ll focus on the security capabilities of WCF RIA Services. Security is one of the areas where RIA Services took something that is vitally important to an application and can be very complex when using WCF on its own, and gave us a simple to use approach that covers 80% of the cases with minimal code and confusion, while still covering most of the remaining 20% nicely with the extensibility hooks in RIA Services for security. Like many other aspects of RIA Services, you are insulated from the service level plumbing that needs to happen to secure your communications and calls and just lets you focus on putting the checks and balances in place to make sure that only things that are supposed to happen in your application are allowed.

There are three concerns to focus on with service security: authentication, authorization, and transfer security. Authentication is simply the act of identifying the caller and determining if you believe they are who they say they are. Authorization is determining what you are going to allow them to do once you know who they are. And transfer security has to do with protecting the messages when they hit the wire so that someone cannot view or tamper with the messages enroute. In addition, you have client side security concerns in that you may not want anyone to be able to run your Silverlight application, and when they do, you want to know who they are both so you can make the service calls and to authorize different functionality on the client side. RIA Services has built-in functionality to cover all of these based on proven and standardized protocols and based on the pre-existing infrastructure of .NET for security.

On important thing to understand is that the security model of the service calls from your Silverlight client to your domain services is a separate thing from the security provided by the hosting web application. When your Silverlight application launches initially in the browser, the host web page is first accessed from the site. Then your XAP is downloaded as a file and launched on the client side. After it launches, the application can make RIA Service calls to the back end, and those calls first manifest themselves on the server side as HTTP requests for an svc file to the web server. This process is shown in the figure below.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/9-10-2010%202-15-37%20PM_2.png)

The security of those calls is determined by the configuration of the hosting web site, which may demand Windows or Forms authentication itself to restrict access to the files. You could choose to rely entirely on site level security, but often you need to have more explicit points of control inside your client and service code where you need to know who the user is and what you are going to allow them to do. This is where RIA Services security steps in and gives you the control you need, both server and client side.

WCF RIA Services security allows you to:

* Leverage membership and role providers configured at the host site to do the authentication and authorization look ups (as well as tie into the Profile system if desired)
* Use Windows or Username/password credentials out of the box, or other credentials using the extensibility hooks
* Declaratively perform authentication and authorization checks on the server side
* Have a client side security context for knowing if the user is authenticated, what their identity is, and what roles they are associated with
* Perform custom authentication and authorization steps on client or server side through the extensibility hooks
* Enforce protection of the messages through SSL

The really quick way to start an application to use WCF RIA Services security is to use the Silverlight Business Application project template. That template sets up all the configuration and services stuff that I am going to walk through in this chapter by default. But I don’t think you should use any capability without understanding what it is doing for you, especially with something as important as security. So I am going to walk through the mechanisms to set it up manually so you understand what you need and what the different pieces do for you.

So let’s dive right in. You can [get the completed sample code for this chapter here](source_code/TaskManagerPart7.zip).

### Authenticating the User

The most common scenarios for authenticating the user include using either Windows credentials or using a username/password (username credentials). For the Windows case, you may require the user to enter a domain account on the client side because they could be working from a home or public computer that is not part of the domain, or you might leverage integrated security to have the hosting browser or application send the Windows credentials automatically. For the username case, you may be validating the username against a database, LDAP or other custom store. In both of these cases as long as you don’t need to do anything too fancy, it can be extremely simple to set up by letting the host web site and browser do the heavy lifting for you. You could choose to authenticate the user in an ASP.NET web page to establish a forms authentication session before you launch the Silverlight application. Doing so would let you secure access to the XAP file download as well as the svc file service calls just based on web site security without doing anything inside of your Silverlight app. Or you could allow unrestricted XAP file download and launch, and then authenticate the user after the Silverlight app launches. There are many options supported by Silverlight itself here, regardless of whether you are using WCF RIA Services or not.

On top of the basic security mechanisms of the host site and Silverlight, you may want to have specific points of enforcement in the service calls on the back end. Additionally, you often need to know who the authenticated user is on the client side, as well as what roles they are in, to modify the behavior of the client application appropriately (i.e. hide or disable commands that the user is not authorized to invoke). Natively in Silverlight, there is no security context, even if the application launch happens as part of a secure session in the browser. The threads do not have security principals on them like in a normal .NET application that could tell you who the authenticated user is. WCF RIA Services steps in here for us and provides the infrastructure to not only authenticate the calls on the server side, but to also return the authentication and authorization context to the client so that you can easily address these scenarios.

If you don’t need the client to know about the user and their roles, it is really simple to secure the back end with RIA Services. The first step is to turn on authentication in the web host site with either Windows or Forms as the mode.

<authentication mode="Forms"/>

Once you have done this, RIA Services can use the configured or default membership provider to look up the username/password that is sent from the client to authenticate them. If you choose to use Windows credentials, those credentials would be validated by IIS itself through the OS. You can collect the user credentials in either case before the Silverlight application launches through a web login form that uses normal Forms Authentication to pass the user credentials in a cookie, or by using RIA Services to pass the credentials from within the Silverlight application. There is nothing special about configuring membership or role providers with respect to RIA Services. That topic has been well covered in many other places in the context of ASP.NET security or Silverlight itself. For a good walkthrough of how to configure things, see the [configuration section of this walkthrough](http://msdn.microsoft.com/en-us/library/879kf95c(VS.80).aspx).

For this chapter, I am going to assume you want to provide the login experience as part of the Silverlight application, and that you will want the user identity and roles available on the client. That means you will need to expose the XAP file from the hosting web site for unauthenticated download unless you want to force the user to log in twice. If instead you want to authenticate via a web page AND you do not need to do any client side authorization or user customization, then you could just rely entirely on the server configuration and services code. But the richer scenarios enabled by RIA Services is what I want to focus on.

#### Step 1: Set up your Membership Provider

To minimize the amount of configuration required to run the sample code, I am using a custom membership provider that is part of the web host project. That provider works against the User table that is part of the TaskManager database. Additionally, that provider just expects the passwords to be stored in the clear in the database so that you can use a predefined account in the database (Username=Brian, Password=IDesign) with no set up. Naturally this is not what you should do in production. With a minor modification to the membership provider, you can hash the incoming password and compare to a hashed version in the database. This requires you to have a mechanism for creating the user accounts that will use the same hashing algorithm when creating the user account. Again, there are lots of examples out there of doing this, so I won’t cover that here.

The custom membership provider just uses the same Entity Framework model to look up username/password combinations to authenticate the calls. The custom membership provider looks like this:

public class CustomMembershipProvider : MembershipProvider

{

public override bool ValidateUser(string username, string password)

{

using (TaskManagerEntities context = new TaskManagerEntities())

{

var user = context.Users.Where(u => u.Username == username &&

u.Password == password).FirstOrDefault();

return user != null;

}

}

public override string ApplicationName

{

get { return "TaskManager"; }

set { }

}

// Other overrides not implemented

...

}

At runtime, the only method in the provider that matters is the ValidateUser method. It will look up the credentials passed in and validate them. If valid, the membership provider and RIA Services will set up an authentication context on the server side. If you expose things right from your services, you can get that authentication context back in the client.

After you have defined your custom provider, or if you want to use the built-in SQL provider, you just need to configure the site for that provider:

<system.web>

<authentication mode="Forms" />

<membership defaultProvider="myCustomProvider">

<providers>

<add name="myCustomProvider" type="TaskManager.Web.CustomMembershipProvider,TaskManager.Web"/>

</providers>

</membership>

...

</system.web>

Now your site and RIA Services have enough information server side to authenticate the user with a username/password. However, you need a little more support client side to provide that information.

#### Step 2: Create an Authentication Domain Service

In order to have the right support on the client side to send the credentials and to establish a client side authentication context after successfully authenticating, you need to define an Authentication Domain Service as part of your host site. This is really a trivial matter unless you want to get into custom authentication scenarios because the RIA Services base classes provide you with everything you need for authenticating and authorizing through the membership and role providers.

To do this, just go to the host web site project, and add a new item to the project (right click on the project in Solution Explorer and select Add > New Item). From the Web category, select the Authentication Domain Service template. Name it **TasksAuthenitcationDomainService**. A class similar to the following will be added. The only modifications I made here are to remove some comments and change the name of the user class used by this service so it does not conflict in name with the one that is defined as part of our Entity Framework model.

[EnableClientAccess]

public class TasksAuthenticationDomainService : AuthenticationBase<AuthUser>

{

}

public class AuthUser : UserBase

{

}

The **AuthenticationBase** class takes care of calling into the membership provider when the user logs in. Because this class is a domain service, a client domain context will be generated and gets tied in automatically with the other domain contexts so that they can secure their calls based on this authentication service as well. The **UserBase** class provides the basic information about an authenticated user such as the identity name, roles and so on. You can derive from this to add any user specific properties that you want to associate with the authenticated user. The authentication service will return an instance of that user type to the client side after successful authentication so that it provides the full context of who the user is and what they can do, in addition to whatever other user specific information you want to add on.

#### Step 3: Create a WebContext instance for use on the client side

Part of the client generated code includes a class called **WebContext**. This class gets enhanced after adding your authentication service to include a property called **User** of the **AuthUser** type used by the authentication service. The base class exposes methods for **Login** and other authentication and authorization related functionality. Even though this type gets defined through the code generation on the client side, you need to initialize the web context for the client and set its authentication type. To initialize the web context, you add an instance of it to the **ApplicationLifetimeObjects** collection of the **App** class in the **App** constructor after setting its **Authentication** property:

public App()

{

this.Startup += this.Application\_Startup;

this.Exit += this.Application\_Exit;

this.UnhandledException += this.Application\_UnhandledException;

InitializeComponent();

WebContext context = new WebContext();

context.Authentication = new FormsAuthentication();

ApplicationLifetimeObjects.Add(context);

}

#### Step 4: Login the user

From there it is up to you to decide when and where to authenticate the user in the client application. But if you are going to secure the services, it needs to be before you start calling the domain services. In the TaskManager application, I went with the simplest, crudest approach. As the **MainPage** loads, it pops a **ChildWindow** derived pop up **LoginForm** that lets the user log in. If log in is successful, the tasks view is loaded. I factored out the XAML that was previously part of the **MainPage** markup into a separate user control called **TasksView** that I load into a **ContentControl** in the **MainPage** if log in is successful to defer the calls to the services until after login is successful. The code in the **MainPage** code behind now looks like this:

public partial class MainPage : UserControl

{

public MainPage()

{

InitializeComponent();

Loaded += OnLoaded;

}

private void OnLoaded(object sender, RoutedEventArgs e)

{

if (!WebContext.Current.Authentication.User.Identity.IsAuthenticated)

{

LoginForm login = new LoginForm();

login.Closed += (s, e2) =>

{

TasksView view = new TasksView();

MainContent.Content = view;

};

login.Show();

}

}

The **LoginForm** ChildWindow code looks like the following

public partial class LoginForm : ChildWindow

{

public LoginForm()

{

InitializeComponent();

}

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

{

LoginOperation loginOp = WebContext.Current.Authentication.Login(

new LoginParameters(UsernameTextBox.Text, PasswordTextBox.Text));

loginOp.Completed += (s2, e2) =>

{

if (loginOp.HasError)

{

errorTextBlock.Text = loginOp.Error.Message;

loginOp.MarkErrorAsHandled();

return;

}

else if (!loginOp.LoginSuccess)

{

errorTextBlock.Text = "Login failed.";

return;

}

else

{

errorTextBlock.Text = string.Empty;

DialogResult = true;

}

};

}

}

The **LoginForm** code calls the Login method on the **WebContext**.**Authentication** property that was initialized in the **App** class. The user can only get past the login dialog if they successfully authenticate. If their credentials are invalid, the **Login** call will succeed, but the **LoginSuccess** property on the **LoginOperation** result will be set to false.

With this in place, you now have everything you need for end-to-end authentication and authorization in the client and services.

#### Step 5: Secure the services

All it takes to protect your domain services from unauthenticated users now is a single attribute: **[RequiresAuthentication]**. This attribute can be placed on the domain service class and all exposed operations on the class (Query, Insert, Update, Delete, and Invoke methods) will be protected. You can also place it at the method level if there are only particular methods you want to require authentication for. For example, in a product catalog scenario, you might allow anyone to query the catalog, but only specific users can modify the catalog.

For the **TasksDomainService**, I am securing all calls:

[EnableClientAccess()]

[RequiresAuthentication]

public class TasksDomainService : LinqToEntitiesDomainService<TaskManagerEntities>

{ }

A quick way to prove to yourself that the authentication is working is to override the Initialize method in your domain service and check the user identity on the context. You can also capture the authenticated user and use it a moment later when the domain service method is called.

private IPrincipal \_User;

public override void Initialize(DomainServiceContext context)

{

base.Initialize(context);

Debug.WriteLine(context.User.Identity.Name);

\_User = context.User;

}

### 

### Authorization with RIA Services

Once you have verified who the user is, you may want to make decisions about what they can do, both server side and client side. To do so, you need to enable the role provider on the server side and have a configured role provider that will answer questions like “what roles is this user in” and “is this user in this role”. Then you can make role assertions on the server and client side and either prevent the user from executing some logic if they are not in the right role, filter data based on them not being in a particular role, or modify the UI based on their role.

#### Step 1: Define the role provider you want to use

Like the membership provider, you can use the default SQL Server provider, a Windows provider that is in the framework, or write your own custom provider. For the sample application, I wrote a simple custom provider. Like the membership provider, there is only one method you have to implement to support authorization at runtime, **GetRolesForUser**:

public class CustomRoleProvider : RoleProvider

{

public override string[] GetRolesForUser(string username)

{

if (username == "Brian") return new string[] { "Manager" };

else return new string[]{};

}

public override string ApplicationName

{

get { return "TaskManager"; }

set { }

}

// Other overrides not implemented

...

}

#### Step 2: Enable the role manager and configure the provider

In your web.config file for the domain services host, ensure the role manager is enabled and your provider is specified if not the default:

<roleManager enabled="true" defaultProvider="myCustomProvider">

<providers>

<add name="myCustomProvider" type="TaskManager.Web.CustomRoleProvider,TaskManager.Web"/>

</providers>

</roleManager>

#### Step 3: Demand roles in domain service methods

To ensure that only users in a certain role are allowed to execute a particular domain service method, you use the **[RequiresRole]** attribute. It takes a **params string[]** of role names. For example, if only managers are allowed to add new customers, I can enforce that in the domain service like so:

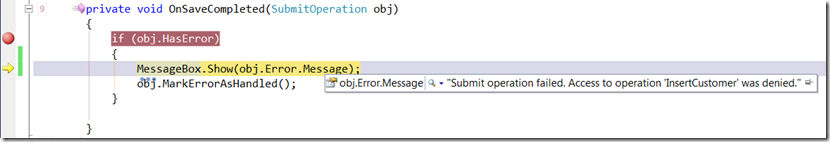
[RequiresRole("Manager")]

public void InsertCustomer(Customer customer)

{

}

If you then add a customer on the client side and you are not in the Manager role, when SubmitChanges is called on the domain context, you will get an access denied exception:

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/9-14-2010%208-02-53%20AM_2.png)

#### Step 4: Filter data based on roles or the authenticated user

You might also need to filter what data is exposed based on who the user is or what role they are in. In the sample application, I have the requirement that users in the Manager role can see all **Tasks**, but other users can only see the tasks for which they are in the associated users collection (many-to-many relationship between **Tasks** and **Users** at the database).

To do this, I can use roles and user identity inside my domain service methods. The authenticated user was captured in the **Initialize** method described in the previous section. I can now modify the **GetTasks** method like so:

public IQueryable<Task> GetTasks()

{

if (\_User.IsInRole("Manager"))

return this.ObjectContext.Tasks.Include("TimeEntries").

Include("Customer").OrderBy(t => t.StartDate);

else

{

return ObjectContext.Tasks.Include("TimeEntries").Include("Customer").

Where(t=> t.Users.Where(u=>u.Username == \_User.Identity.Name).FirstOrDefault() != null).

OrderBy(t=>t.StartDate);

}

}

You can see that if the user’s role is Manager, all Tasks are retrieved as before (possibly filtered by the client side expression tree sent when the method is called), but if the user is not in the Manager role, the results are filtered server side to only return the Tasks for which they are linked as a user on that Task.

#### Step 5: Make client-side authorization decisions

If the user is not a Manager, then the Add Customer button on the client side should never be enabled in the first place (or possibly hidden) so that the access denied exception is never reached. To do this, you need to check the user’s roles on the client side. This is easy to do through the **WebContext** since its **User** property will be populated after authentication is complete.

The view model previously had a **CanExecute** handler for the **AddCustomerCommand** that was mistakenly driving enablement off of the selection of a **Task**. The adding of a customer is actually decoupled from the task selection, but now I want the command to be disabled if the user is not a manager. So I simply updated the **CanExecute** handler for the **AddCustomerCommand** in the view model to the following:

private bool OnCanAddCustomer(object arg)

{

if (WebContext.Current.User.IsInRole("Manager"))

return true;

else

return false;

}

Since the authentication is complete before the view and view model load, the **WebContext** will already know who the user is and have the roles the user is associated with populated by the back end.

### Requiring Message Protection

In the sample application, the data and credentials are all being passed in clear text when the messages go back and forth from the client to the server and back. Obviously that is not a good idea for security. When you are securing your site and using WCF RIA Services, you really need to use SSL to protect the messages. To enforce that the service does not get deployed without the protection of SSL, you can simple add a property to the [**EnableClientAccess**] attribute in your domain services:

[EnableClientAccess(RequiresSecureEndpoint=true)]

### Summary

In this chapter I have walked you through the core security features of WCF RIA Services that allow you to easily authenticate the user and authorize what actions they can take in the client and in the services. There are a number of additional things you can do including performing more complex authentication inside your authentication service and defining custom authorization attributes.

If you want more explicit control over the authentication process inside of your services, you can override the **ValidateUser** method on the **AuthenticationDomainService** derived class you add to your server project. Then instead of doing what the base class does – call out to the configured membership provider – you are in complete control inside of your service to do whatever lookup of the client credentials you need. The advantage of the membership provider approach is that the provider you write becomes reusable not only in a WCF RIA Services application, but can also be used with ASP.NET, WCF, and even WPF and Windows Forms clients through Client Application Services. The advantage to bringing it inside your domain services is that you can more tightly integrate the user object used for authentication and a domain object like the separate User type in the Tasks Entity Framework model. For a good article and example of that, see the BookClub RIA Services sample and the [authentication](http://www.nikhilk.net/RIAServices-Authentication.aspx) and [authorization](http://www.nikhilk.net/RIAServices-Authorization.aspx) posts by Nikhil Kothari.

[Sample code](source_code/TaskManagerPart7.zip)

# Chapter 8: Testing and Debugging

### 

### Introduction

In this chapter, I am going to focus on two aspects that are just as important as how to write the code – how to write the code in such a way as to be more testable, and how to debug and diagnose when your code is not doing exactly what you expect. On the testing front, I am mainly focusing on the ability to unit test important parts of your code, such as logic in a view model that consumes a domain service. I’ll also talk briefly about how to test on the server side, although full coverage of that will have to wait for another chapter. Along the way, you will learn a lot more about the magic that is going on under the covers of the domain context as well.

On the debugging front I will discuss what mechanisms and techniques you can use to understand what is going on outside of your code in the RIA Services stack on the client and server side as well as what is happening at the wire level to try to infer the cause when you are not getting the results you expect from your RIA Services calls.

You can [get the code for this chapter here](source_code/TaskManagerPart8.zip).

### 

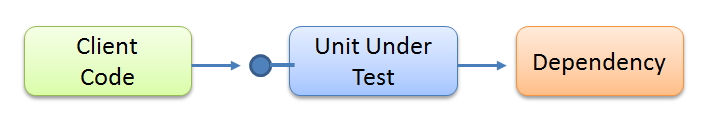
### Unit Testability

With respect to unit testing, I am assuming you are familiar with the general concepts of unit testing and separation of concerns. If you have never tried to write unit tests, you are not going to learn the fundamentals here. And in fact, you are going to be thrown to the bottom of the deep end if you try to start here, because unit testing RIA Services code is not for the faint of heart. One of the downsides of the WCF RIA Services framework is that it is what I would call “test-resistant" code. The framework does a lot of work under the covers for you that gets intimately intertwined with the code you write that you want to test. It makes it hard to write unit tests that just exercise your logic code.

If you are new to unit testing, I’d strongly recommend the book [The Art of Unit Testing by Roy Osherove](http://artofunittesting.com/) as a great starting point for learning about unit testing concepts and techniques. Additionally, you need to understand how the Silverlight Unit Test framework works, including how to test asynchronous execution in that environment. I’ll explain some of the latter as I go, but some good pre-reading is the [Silverlight Unit Test tutorial](http://www.jeff.wilcox.name/2008/03/silverlight2-unit-testing/) and this post on [asynchronous testing by Jonas Folleso](http://jonas.follesoe.no/PermaLink,guid,f2fc834b-704e-40ae-9684-44cfa8096bed.aspx).

I am also assuming you understand the motivations and reasons for separating the logic of a view from the structure of the view – specifically the Model-View-ViewModel pattern that I showed in chapter 4 of the ebook, or possibly the Model-View-Controller or Model-View-Presenter patterns. If you don’t separate your logic from the view definition, you are going to have a hard time unit testing it in the first place because the controls and elements in the view often expect that they are running in a UI context and will not have the same behavior if they are not being rendered. So the assumption here is that on the client side you are using the MVVM pattern.

The main thing I am going to focus on in this chapter has to do with the ability to unit test code in your view models if using a domain context in the view model, as well as the ability to unit test business logic that is executed in your domain service. In either case, the key concept when unit testing is the need to mock out calls to external dependencies. In the figure below, the unit under test is typically a chunk of logic code such as that found in a view model in the presentation layer or a business logic component on the server. The client code is whatever calls the exposed API directly. In the case of a view model, it may be the view or it may be some of the client hosting code that instantiates the view model, such as an application controller. In the case of a server business logic component, the calling code may be the WCF RIA Services infrastructure itself if it is something like a domain service method, or it could be a child component that gets called from your update methods within your domain service. The logic you are trying to test often has dependencies of its own, parts of the code that call out to downstream components that go outside the local execution scope themselves. Examples would be data access components on the server side or the service calls being made under the covers of the domain context on the client side.



In order to unit test that code in the middle, you need to be able to replace the client code with your unit test code, and you need to be able to substitute a mock or stub implementation of the dependency so that the unit under test calls the mock instead of the real dependency. You also need a way to specify the behavior of that mock object so that it behaves as the real dependency would in terms of inputs and outputs in the context of each unit test you are going to write against the unit under test. You can do this by writing “manual mocks” or classes that implement the same interface as the dependency but that have their behavior coded into the test environment or by using a mocking framework such as [MOQ](http://code.google.com/p/moq/), [Rhino Mocks](http://www.ayende.com/Wiki/Default.aspx?Page=Rhino+Mocks&AspxAutoDetectCookieSupport=1), [TypeMock Isolator](http://www.typemock.com/), or [JustMock](http://www.telerik.com/products/mocking.aspx). For this chapter, I’ll be using MOQ.

### 

### Unit Testing the View Model or Client-Side Service That Uses a DomainContext

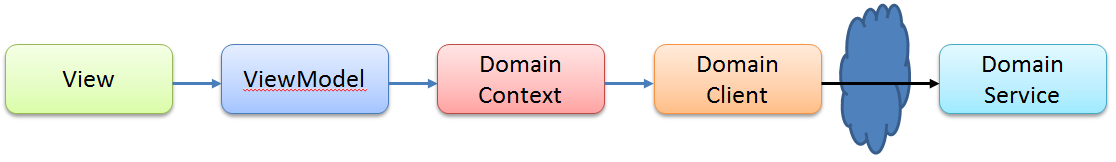
There are a couple places in a Silverlight client architecture where you may create and use **DomainContext** objects. If you are following the MVVM pattern and separating the concerns of your application, the two most likely places you would be creating and making calls through a domain context is in a view model or a client side service (chunk of re-usable code obtained through [service locator pattern](http://msdn.microsoft.com/en-us/library/ff648968.aspx)). In either case, if the surrounding code has any complexity or logic surrounding those service calls and the manipulation of the entities, you should really be writing some unit tests as you write the code to help flesh out that behavior and verify its correctness. The trick is that the domain context makes asynchronous service calls you when make your Load, **SubmitChanges** and Invoke method calls. And in a unit test environment, you don’t want those calls to actually make service calls, you want to be able to mock them out.

You might be tempted, as I was, to say “no problem, just put an interface-based wrapper around the domain context object and mock that.” While that is actually fairly easy to do, it doesn’t actually make the problem much easier. The real challenge about unit testing code that is using a domain context as a dependency has to do with both the async API exposed, and the state management that the domain context does behind the scenes maintaining the collections of entities and the change tracking on those entities. Also things like associating errors in the service calls with the entities for validation purpose, etc. Bottom line, that little domain context is working its butt off for you, and if you try to mock out its public API, your mocking code will have to do a ton of work to simulate the same behavior. Since the domain context really becomes part of the functionality of your view model, you won’t really be testing the functionality of your view model anymore, you will just end up testing that your mock is doing what you told it to, which is pointless.

Dealing with the async nature of the domain context calls in your unit tests can be made much easier if you can transform the async API into single synchronous method calls for the purposes of mocking them in your unit tests. Mocking a synchronous API is easier because the outcome of any one call to a dependent object can be mocked as being complete when that method or property call completes. Most mocking frameworks are designed around mocking synchronous calls as well.

It turns out that there is a better place to do the mocking of the service calls rather than trying to do it on the asynchronous surface of the domain context. Under the covers of the domain context object, there is a **DomainClient** object. This is the object that actually makes the real service calls that hit the wire. Those service calls are still made based on an async API at the domain client level, but if you mock out the domain client, it is fairly easy to turn the async calls into synchronous calls at the point where they are dispatched. Then in the unit test environment, you can mock out the synchronous calls so that the unit code is easier to write and it runs more synchronously. You’ll see what I mean by this in the code shortly.

The call chain when you call through the domain context looks like the following figure:



For example, consider what happens when you do a Load operation on a domain context from your view model. The domain context takes the **EntityQuery<T>** that you passed into the Load method and passes it to a base class as an **EntityQuery**, a type not tied to a specific entity type. The base class makes an asynchronous call into the domain client to a method called **BeginQueryCore**. When completion of that async operation is indicated by the domain client through a callback to the domain context (normally when the web response is received in the real domain client used by the RIA Services code), the domain context then calls **EndQueryCore** and gets back a  **QueryCompletedResult** that contains the retrieved entities in a general purpose container that can contain any kind of entity. The contained entities are then unpacked from that payload by the domain context and it updates the entity collections exposed from the domain context. It then calls and Load completion callbacks or event handlers.

To mock out the calls to the service then, you just need to replace the domain client that your domain context would use by default and replace it with one where you can mock the calls to the server in a synchronous way. The approach I will be demonstrating in this part of the chapter is based on some guidance put out by Nikhil Kothari from the RIA Services product team on his blog [in this post](http://www.nikhilk.net/NET-RIA-Services-ViewModel-Pattern-2.aspx). That post doesn’t take it very far, only showing a simple query. So I am picking up where that left off and showing how to handle a little more complexity in terms of verifying queries as well as handling **SubmitChanges** calls.

The first step is to create a derived class from the **DomainClient** class that turns the async calls to the server into synchronous method calls that you can mock out more easily. Once you get down to the domain client level, there are really only three logical operations that are being dispatched on the wire by a domain client – Query, Submit, and Invoke. Basically, the **DomainContext** base class effectively translates between the type specific API exposed by the derived **DomainContext** class that gets code generated on the client for each domain service (i.e. **TasksDomainContext**)  and the more general purpose API exposed by the **DomainClient** base class. It doesn’t throw away all type safety, but wraps the specific entity types in generic wrappers and containers that get passed at the wire level (**EntityQuery** and **EntityChangeSet** for Query and Submit respectively).

### Creating a Mock DomainClient

So if you want to mock out the domain client and do so with a blocking implementation that is easier to work with in unit tests, you just need to add a synchronous API to the **DomainClient** base class that gets called in a blocking way from the Begin/End method pairs associated with Query, Submit, and Invoke.

The following class shows the **TestDomainClient** class that I put together for this chapter. It handles the Query and Submit parts, but I’ll leave it as an exercise for the reader (or for me in a future post) to flesh out and demonstrate the Invoke operations. The first step is to derive from **DomainClient** and add a set of synchronous methods that can be mocked out in the unit tests.

public abstract class TestDomainClient : DomainClient

{

private SynchronizationContext syncContext;

protected TestDomainClient()

{

this.syncContext = SynchronizationContext.Current;

}

public abstract IQueryable<Entity> Query(EntityQuery query);

public abstract IEnumerable<ChangeSetEntry> Submit(EntityChangeSet changeSet);

public abstract InvokeCompletedResult Invoke(InvokeArgs invokeArgs);

protected override sealed IAsyncResult BeginInvokeCore(InvokeArgs invokeArgs,

AsyncCallback callback, object userState) {}

protected override sealed InvokeCompletedResult EndInvokeCore(IAsyncResult asyncResult) {}

protected override sealed IAsyncResult BeginQueryCore(EntityQuery query,

AsyncCallback callback, object userState) {}

protected override sealed QueryCompletedResult EndQueryCore(IAsyncResult asyncResult) {}

protected override sealed IAsyncResult BeginSubmitCore(EntityChangeSet changeSet,

AsyncCallback callback, object userState) {}

protected override sealed SubmitCompletedResult EndSubmitCore(IAsyncResult asyncResult) {}

}

The overrides at the bottom of the class show the inherent API of the base class that I need to override. The three abstract synchronous methods at the top are the ones that I want to be able to mock out in the unit tests. The **SynchronizationContext** stuff at the top is a little necessity to deal with thread dispatching on the UI thread since this code will run in the Silverlight Unit Test framework, which runs on a UI thread itself.

The trick here is to just make the async methods call the new synchronous abstract ones and cause the async API (which will be called by the domain context) to return the results specified by the synchronous methods. The implementation for the Query side of things looks like this:

protected override sealed IAsyncResult BeginQueryCore(EntityQuery query, AsyncCallback callback, object userState)

{

var results = this.Query(query);

var asyncResult = new TestAsyncResult

{

Callback = callback, AsyncState = userState,

Entities = results.ToArray(), TotalCount = results.Count()

};

this.syncContext.Post( cb => ((TestAsyncResult)cb).Complete(), asyncResult);

return asyncResult;

}

protected override sealed QueryCompletedResult EndQueryCore(IAsyncResult asyncResult)

{

var localAsyncResult = (TestAsyncResult)asyncResult;

return

new QueryCompletedResult(

localAsyncResult.Entities,

localAsyncResult.IncludedEntities,

localAsyncResult.TotalCount,

new ValidationResult[0]);

}

You can see that what the code does is call the synchronous abstract method in the **BeginQueryCore** method with the **EntityQuery** argument that was passed down from the domain context to come up with the result synchronously. Then it puts together a custom **IAsyncResult** object that packages up the results with a structure that will be passed to the End method when it gets called by the domain context. That call is triggered by invoking the callback object that was passed into the **BeginQueryCore** method, which is done through a Post call on the **SynchronizationContext** to make sure it gets dispatched to the UI thread if needed. The **TestAsyncResult** class is just a simple data structure that contains the normal members of **IAsyncResult** plus some extra properties to package up the different data structures needed to dispatch results from the Begin method to the End method for the domain client operations. You can see the full code of the **TestDomainClient** and the **TestAsyncResult** class in the [source code for this chapter](source_code/TaskManagerPart8.zip).

The **EndQueryCore** method then just takes the values that were packaged up on the custom **IAsyncResult** object and uses them to populate the **QueryCompletedResult** object that is returned to the domain context.

### Testing a View Model Command Handler for a Query

With code like that in place, you can then start writing unit test on top of your view model code that makes a **Load** call. For example, consider this simple command handler in the **TasksViewModel** class:

internal void OnSearchByDate(object param)

{

\_Context.Tasks.Clear();

EntityQuery<Task> query = \_Context.GetTasksQuery();

LoadOperation<Task> loadOp = \_Context.Load(

query.Where(t => t.StartDate >= LowerSearchDate && t.StartDate <= UpperSearchDate));

}

There is not much logic here to warrant a unit test in terms of the view model code itself, but you might want to write one that verifies that after executing a search, the Tasks collection exposed by your view model contains the results of that search, and also verify that it looks like it executed the query by filtering on the **StartDate** property of the tasks.

The unit test code to do so looks like the following:

[TestMethod]

[Asynchronous]

public void OnSearchByDate\_RepopulatesTasksByStartDate()

{

// Arrange ---------------------------------

bool loadExecuted = false;

var domainClientStub = new Mock<TestDomainClient>();

domainClientStub.Setup(dc => dc.Query(It.Is<EntityQuery>(query => query.EntityType == typeof(Task))))

.Callback(() => loadExecuted = true) // used to verify it was called

.Returns(() => new Entity[] // dummy data

{

new Task { TaskId = 42, TaskName = "Task 1" },

new Task { TaskId = 43, TaskName = "Task 2" }

}.AsQueryable()

);

// Act -------------------------------------

TasksViewModel viewModel = new TasksViewModel(domainClientStub.Object);

viewModel.OnSearchByDate(null);

// Assert ---------------------------------

Assert.IsTrue(loadExecuted); // did it execute?

EnqueueConditional(() =>

viewModel.Tasks != null &&

viewModel.Tasks.Count() == 2); // Did we get the right results

domainClientStub.Verify(dc =>

dc.Query(It.Is<EntityQuery>(query =>

query.Query.ToString().Contains("StartDate") // Did it query by the right stuff

)));

EnqueueCallback(() => // Did it populate the view model

{

Assert.AreEqual("Task 1", viewModel.Tasks.ToList()[0].TaskName);

Assert.AreEqual("Task 2", viewModel.Tasks.ToList()[1].TaskName);

});

EnqueueTestComplete();

}

So the first thing you might notice is that it took about 30 lines of actual test code to exercise 4 lines of actual view model code. That is not going to make you or your manager happy. Even though this is only exercising 4 lines of your code, it is exercising a whole lot more than that under the covers in the RIA Services stack. That probably raises the question of whether you should be writing tests that are really just testing that the RIA Services code is doing what it should. Generally the answer to that would be no. But if you want to have a test to verify that your view model properly executes a search, this is what you are going to have to go through.

If the method were as simple as the one I am showing above, I’m not sure I would be able to convince myself that the test coverage is worth it. However, if you have a more complicated interaction with the data after executing the search, including some post processing of the results with surrounding conditional logic or computation on the client side, then you should probably be testing that. The same test pattern shown above would cover you for making sure you are in control of the query part of it in your unit test, and then you would just add whatever verifications are needed to make sure the real logic of your view model is correct. However, using such a simple chunk of view model code here lets us focus on what is needed to do the testing in general without getting mired down in the actual logic verification itself.

Let me walk through the code above line by line to make sure it is clear what all the ceremony is about. The first thing the test does is to create a mock implementation of the **TestDomainClient** base class using the MOQ framework. It then uses MOQ’s API to express an “expectation” of how that mock object will behave. Specifically, lines 8-15 basically say the following: “I expect that under the covers of the method call I am about to make, my mock domain client object will have its **Query** method called. I expect that it will be called with an **EntityQuery** for **Task** objects. When it gets called, I want to set a flag (**loadExecuted**) so that I can verify that it was called. Additionally, I need that method to return these two instances of **Task** objects as the resulting entity collection when it completes.” That is probably the most complicated part of the test itself, and most of the complexity just has to do with understanding the lambda-based syntax of saying that in code to MOQ.

After that, the code is done with the “Arrange” phase of the Arrange-Act-Assert pattern of unit testing and is ready to execute the target method, which it does in a straightforward call to the view model object. Note that the code uses manual dependency injection to pass the mock domain client to the view model. This required adding an internal constructor to accommodate that and some clean up of the main constructor’s code to factor out the common construction pieces (i.e. setting up the command and initializing member variables) so that it could be called from both the default constructor that uses the default constructor of **TasksDomainContext** and the following one that accommodates dependency injection in the unit tests.

The injection constructor looks like this in the view model:

internal TasksViewModel(DomainClient client)

{

\_Context = new TasksDomainContext(client);

CommonConstruction();

}

You can see that the magic hook here is that there is an overloaded constructor on the domain context that takes a **DomainClient** instance. This gives me the path I need to inject my mock instance under the covers of the real **TasksDomainContext** that is being used by the view model.

In the Assert phase of the test, I am actually checking four things. Technically this is a bit of a style violation for unit tests – a unit test should test one outcome. In this test I am trying to validate is that the query ran and returned the expected outcome. To ensure that, I am checking it in several ways to show things you could do. The first thing in line 22 is just verifying the flag got set that tells us the mock **Query** method ran. The next in lines 24-26 is an **EnqueueConditional** call to wait until the Tasks collection of my view model has been populated. Because the Load happens asynchronously, this is essential because the synchronous call chain of calling the search method will complete, and then the thread dispatching happens to marshal the results back to the UI thread and then the Tasks will show up in the collection. **EnqueueConditional**allows you to wait in your test and the test framework will basically sample that condition for a short period of time to see if it happens. If it doesn’t happen after the framework knows or assumes it should have been called (based on black magic and voodoo that the test framework is doing), then it gives up and fails your test so it doesn’t block forever. If you get past that conditional without a failure, it means that condition was met, which is an implicit assertion of its own.

After that, lines 28-31 use MOQ to verify that some of the low level parameters (specifically the query expression) that got passed to the mock contained what I expected – confirming that it does look like it was filtering on **StartDate**. I could get more fancy with that to really check that it was filtering on upper and lower bounds, but that gets a lot messier. Finally at the end of the test I verify that the tasks in my view model collection are the ones returned from the query, using the **EnqueueCallback** method from the Silverlight Unit Test framework so that the UI can pump any messages in between if necessary.

Now admittedly this is all very artificial because the real work of a query goes on the server, so this is just going to return whatever we tell the mock to return. But it should at least give you a sense of how to go about providing a mock domain client and being able to specify its behavior and verify that it was called as expected.

### Testing a View Model Command Handler for SaveChanges

Now let me show an example of how to handle testing that the Save Changes command on the view model works correctly. The view model command handler method looks like this:

internal void OnSaveChanges(object param)

{

\_Context.SubmitChanges(OnSaveCompleted, null);

}

Again, not enough here that I would normally write a unit test for, but what if you did want to verify that your changes were getting sent to the server at the right time if there were some surrounding conditional logic determining when that call should happen. You would still need to be able to mock out the call to the server and verify if it happened in your unit test.

Here is a test that verifies that when we call **OnSaveChanges** on the view model, that the server gets called with **Submit**:

[TestMethod]

public void SaveChanges\_CallsServerToSubmitChanges()

{

// Arrange ---------------------------------

bool submitExecuted = false;

var domainClientStub = new Mock<TestDomainClient>();

// Need to use the domain context directly to add an item to the collection

TasksDomainContext testContext = new TasksDomainContext(domainClientStub.Object);

Task newTask = new Task { TaskId = 33, TaskName = "Test" };

testContext.Tasks.Add(newTask);

// Get the associated change set to mock the call

EntityChangeSet resultSet = testContext.EntityContainer.GetChanges();

domainClientStub.Setup(dc => dc.Submit(It.Is<EntityChangeSet>(ecs => ecs.AddedEntities.Count == 1)))

.Callback(()=>submitExecuted = true)

.Returns(resultSet.GetChangeSetEntries());

// Act -------------------------------------

TasksViewModel viewModel = new TasksViewModel(testContext);

viewModel.OnSaveChanges(null);

// Assert ---------------------------------

Assert.IsTrue(submitExecuted);

domainClientStub.Verify(dc => dc.Submit(It.Is<EntityChangeSet>(ecs => ecs.AddedEntities.Count == 1)));

}

In this case, you need to access the domain context directly in the test because you need to provide an **EntityChangeSet** to the mock to return. The only way to get one of those (because of internal constructors and **ReadOnlyCollections** exposed from it) is to use the **GetChanges** method on an **EntityCollection**, which is tucked away under your **DomainContext**. So in this case I added another test constructor that lets me inject the domain context after populating it with its mock domain client and accessing its **EntityCollection**. Feels like a little too much intimacy to me, but seemed to be the only way I could find to get the test written.

The final test in the sample code deals with another piece of nastiness that the view model was doing that I had highlighted in an earlier chapter – popping message boxes and child windows directly. When an error occurs on while sending changes to the server, I wanted to alert the user of the problem. So the quick and dirty approach from an earlier chapter was to pop a message box from the view model when the asynchronous completion of **SubmitChanges** happened and the error was detected. In a clean view model world, the view model just manages state, it does not directly render things itself.

So in this version I used a little chunk of functionality from the new [Prism 4 release](http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3453ab2b-2067-41e4-b087-312d8385cf1b&displaylang=en) which helps address this. I’ll be writing some articles on Prism 4 in the near future, but you can take a look at [this post](http://briannoyes.net/2010/11/15/PromptingTheUserFromAViewModelndashPrism4Gems.aspx) for a quick summary of the feature I am using, called interaction requests. This allows the view model to just expose another property of an interface type, and on that interface is an event that the view can handle (with a behavior) to prompt the user. Now the view model is decoupled from the actual rendering which is very important for unit testing, as well as just clean separation of concerns.

The completion code for the **SubmitChanges** call now looks like this:

private void OnSaveCompleted(SubmitOperation obj)

{

if (obj.HasError)

{

SaveFailedNotification.Raise(

new Notification {

Content = obj.Error.Message,

Title = "Error saving changes" });

obj.MarkErrorAsHandled();

}

}

A custom Prism behavior in the view is the normal subscriber to the **Raise** event on the notification object (of type**InteractionRequest<T>**) and it takes care of presenting a templatable **ChildWindow** as a popup – leaving it up to the owner of the view to decide what the right presentation to the user is. Again imagine in a more real application there might be some more view model logic surrounding this that I also want to unit test, and one of the branches I want to verify is that if a server exception comes back, the user gets prompted.

Now I can write a unit test like so to verify that the view model does what it needs to do to prompt the user when a failed **SubmitChanges** call happens:

[TestMethod]

[Asynchronous]

public void SaveChanges\_WithFailure\_NotifiesUser()

{

// Arrange ---------------------------------

bool userNotified = false;

var domainClientStub = new Mock<TestDomainClient>();

TasksDomainContext testContext = new TasksDomainContext(domainClientStub.Object);

Task newTask = new Task { TaskId = 33, TaskName = "Test" };

testContext.Tasks.Add(newTask);

EntityChangeSet resultSet = testContext.EntityContainer.GetChanges();

domainClientStub.Setup(dc => dc.Submit(It.Is<EntityChangeSet>(ecs => ecs.AddedEntities.Count == 1)))

.Throws(new ArgumentException());

// Act -------------------------------------

TasksViewModel viewModel = new TasksViewModel(testContext);

viewModel.SaveFailedNotification.Raised += (s, e) => { userNotified = true; };

EnqueueCallback(() => viewModel.OnSaveChanges(null));

// Assert ---------------------------------

EnqueueConditional(() => userNotified);

EnqueueTestComplete();

}

Since all that really happens at a view model level now to prompt the user is firing the **Raised** event on the **InteractionRequest<T>**object, I can easily subscribe to that in a unit test and verify that it gets raised when an exception is thrown from the service call (**Submit** in this case). Lines 12-13 take care of setting the MOQ expectation that when Submit is called, it will throw an **ArgumentException**.

So hopefully that shows you how you can still use a domain context directly in your view models, but still be able to unit test the code by mocking out the service calls. The downside as you can see is that it is not very clean or easy. But it is at least do-able, and it is also an easily repeatable pattern.

### Unit Testing Domain Service Code

On the server side, things don’t really get any easier when it comes to unit testing your domain service class logic. If you use the **LinqToEntitiesDomainService** class for example, the WCF RIA Services stack is the client of your domain service and does a lot of work in between each individual method call on your domain service class, all as part of a single service operation from the client. Additionally, in that case you will have the Entity Framework **ObjectContext** class as a direct dependency, which itself is inherently not unit testable since it goes out direct to the database, outside your execution context.

If you have discreet chunks of logic being invoked from your domain service methods (i.e. validation logic that gets executed on each entity for each update or insert), that code can usually be factored out into separate methods or components that get called from the domain service operations and it can often be tested on its own. For example, the **TasksDomainService** has this method that gets called for validation on an insert or update operation on a task:

private void ValidateCustomerHours(Task task)

{

if (task.CustomerId.HasValue)

{

int customerId = task.CustomerId.Value;

bool timeAllowed = IsTotalTimeAllowed(customerId, task.TotalTime);

if (!timeAllowed)

throw new ValidationException(

"You cannot enter more than 10 hours per billing cycle for this customer.");

}

}

It would be a trivial task to write a unit test for a method like this. However, if you start weaving logic into the domain service Insert, Update, and Delete operations themselves, you will have a hard time unit testing them because you would need a means to mock out the calls to the Entity Framework **ObjectContext**. Again, using a commercial mocking framework like JustMock or TypeMock Isolator you can do that, but with the free open source ones like MOQ or Rhino Mocks, you generally cannot.

The only choice at that point is to start designing all of your domain services as POCO domain services – ones derived from the **DomainService** base class. This requires you to define your own object model, use the right attributes on your entity properties, and a lot of other concerns that I am going to have to defer until a later chapter.

If you use POCO domain services, then you can factor your data access logic out to a repository pattern, and mock out the repository in your unit tests of the domain service methods. [This walkthrough](http://msdn.microsoft.com/en-us/library/ee707368%28VS.91%29.aspx) in the RIA Services documentation demonstrates this approach.

### Debugging RIA Services

One of the other challenging things about WCF RIA Services at times is that if things are not working as you expect, it is hard to tell what to do to figure it out. The RIA Services stack is doing a lot of work on your behalf. That is the whole point of using RIA Services – they do the work so you don’t have to. Unfortunately, that means there is a lot of magic going on under the covers of a few simple method calls in your client code.

The first step to being able to understand when problems occur is to understand what is going on under the covers of RIA Services. I don’t think anyone should use pre-built plumbing like RIA Services unless they understand what is happening under the covers. Hopefully this and other chapters in this ebook have helped you understand some of what is happening under the covers a little better. Because so much happens under the covers, for debugging you need to be able to peek under those covers and observe what is going on.

The first and most important tip to debugging RIA Services that I have mentioned before is that you should really handle completion of all async calls through your domain service and have code that checks for the **HasError** property to be set on the operation argument. If there is an error and you don’t handle it, it will throw the exception on the client side and it will crash your application. You should have code like this for every completion of every **Load**, **SubmitChanges**, and Invoke method call on your domain context:

private void OnSaveCompleted(SubmitOperation op)

{

if (op.HasError)

{

// Do whatever is appropriate

op.MarkErrorAsHandled();

}

}

Setting a breakpoint here and inspecting the error information is your first line of defense for figuring out what is wrong. As I have already covered in earlier chapters, RIA Services will package up any server processing errors and will raise them here. Since a lot of the server processing happens in the RIA Services stack outside of your domain service method calls, this is often the only place you can set a breakpoint and see the details of those exceptions.

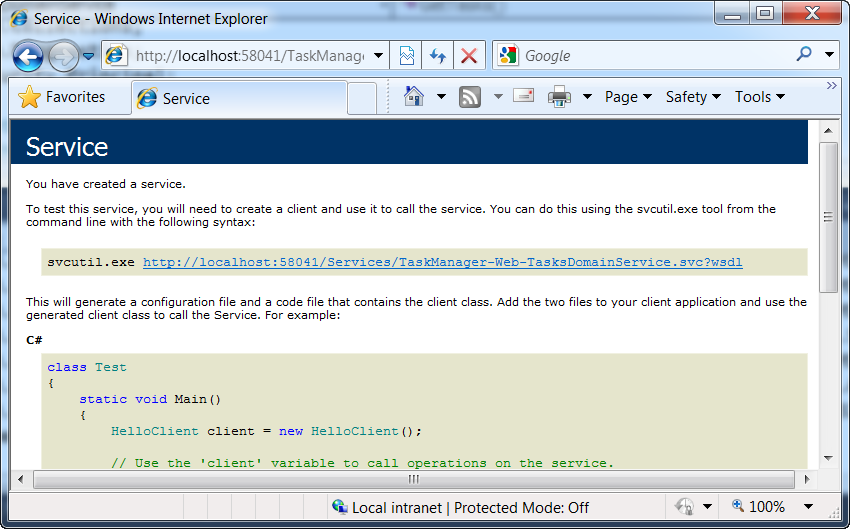
On the server side, another option you have is to override the **OnError** method from your domain service base class. This will be called when there is an exception server side in the process of handling a query operation or in processing a **ChangeSet** for updates. Setting a breakpoint there gives you more direct access to the server stack and variables in your domain service instance while it is actually executing.

It is also important at times to be able to inspect and verify exactly what RIA Services is sending back and forth at the wire level. To do this, you have two tools at your disposal that you should get familiar with. The first is [Fiddler](http://www.fiddler2.com/fiddler2/). This tool lets you inspect the raw HTTP requests and responses on your machine. To use this effectively, you will need to set up your web project that is hosting your Silverlight app and your domain services to run under IIS instead of the ASP.NET Development Server that fires up by default when debugging in Visual Studio. Additionally, it is easier to work with if you set it up so the Silverlight app is launching using your machine name instead of localhost. Setting up and using Fiddler is beyond the scope of this chapter, but you can find some of the key ways to do so [here](http://www.fiddler2.com/fiddler/help/hookup.asp).

Because RIA Services makes binary encoded HTTP REST calls by default, the payloads of the messages may be a little hard to read. You can usually still see the basic structure of the message in there, but it will be a little cryptic. However, if there are errors calling the service, you will see those much more obviously in Fiddler than you will by trying to analyze the error messages coming out on the client side code. And you can usually see enough there to figure out the basics of what is going on.

If that doesn’t work for you, another important tool is the WCF Service Trace Viewer. Since RIA Services is using WCF to make all the service calls, turning on tracing and message logging on the server side will at least give you full insight into what is happening in the WCF stack on the server, and will also give you a chance to inspect the raw messages before they are binary encoded on the wire. For a quick intro on how to turn on tracing and message logging, [check here](http://mkdot.net/blogs/dejan/archive/2008/10/23/wcf-tracing-and-message-logging.aspx).

A final important debugging tip is that if you are getting cryptic errors back from your client domain context calls, one of the first things to check is that the service itself is hosted correctly and that the client is able to call it. The quick check for that is that your domain service is automatically hosted by RIA Services by default at the same site your XAP file is hosted. RIA Services automatically goes looking for the service in the root virtual directory or site and tacks on the fully qualified domain service class name, replacing dots with dashes, and adding a .svc file extension. For example, in the sample code, the domain service is TaskManager.Web.TasksDomainService. The default URL where the service gets hosted by WCF is <rootsite>/TaskManager-Web-TasksDomainService.svc. So when debugging on my local machine hosted in IIS, I can fire up a browser and hit http://<mymachinename>/TaskManager.Web.TasksDomainService.svc and I should see a page like the following:



If there is an error, then I know my domain context is not going to succeed in talking to the domain service and can focus on resolving whatever errors show up here. For a great step through of some of the common errors you will get, see this [Silverlight TV episode with Saurabh Pant](http://channel9.msdn.com/Shows/SilverlightTV/Silverlight-TV-51-Debugging-and-Deploying-WCF-RIA-Services) from the RIA Services team.

### 

### Summary

Testing and debugging are deep topics, and a lot more could be said about each. WCF RIA Services definitely makes it more challenging to unit test your code because to fully leverage what RIA Services can do for you, it tends to get sort of intertwined with your own logic code. So exercising portions of the RIA Services stack as part of your unit tests is necessary, but you still need to be able to mock out the external calls if you want it to be a unit test. This chapter showed you how you can do that on the client side and gave some pointers on how to deal with it on the server side.

For debugging, you need to be able to see what is going on at the wire level a lot of times, and Fiddler or the WCF Service Trace Viewer can let you do that. Additionally, you have a couple of places to hook errors on the client and server sides that were pointed out.

In the next chapter, I’ll talk a little about how to structure your RIA Services applications when you get beyond having just a single client project and single service project.

[Sample code](source_code/TaskManagerPart8.zip)

# Chapter 9: Structuring Your Application

### 

### Introduction

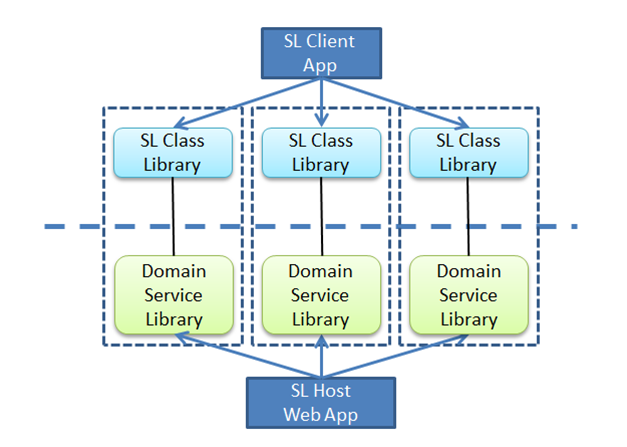
In all the chapters up to this point, I have been dumping all the domain service stuff into the single web project that also hosts the Silverlight application. Additionally, I've been putting a lot of functionality into the single Silverlight Application project on the client side. Granted it is only a couple of views so far. But if this app grew to a couple dozen views, I really would not want to be putting all those into a single project. In a real world app, too much code in one place, whether in a single method, a single class, or a single project is a maintenance liability. Additionally, on the server side you might want to start partitioning the code into logical layers, such as breaking out a separate data access layer project from the domain service code, and you might want separate projects for different sets of domain services.

In this chapter, I'm going to focus on how to break your solution up into multiple projects on both the server and client sides. I'll show how you can break things up into vertical slices that support different use cases in your application or different functional areas. You'll quickly learn how to use WCF RIA Services Class Library projects, as well as how to break things up into multiple XAP files on the client side. Using the Managed Extensibility Framework or [Prism 4](http://www.microsoft.com/downloads/en/details.aspx?FamilyID=3453ab2b-2067-41e4-b087-312d8385cf1b&displaylang=en), you can break up your client application into multiple modules that get developed and built as separate XAP files, and then get downloaded asynchronously when needed by the client application. When you go down this path, unfortunately Visual Studio gets in your way a little bit for setting up your WCF RIA services link with the server project. I'll show you how to easily get past that limitation as well.

You can [get the sample code for this chapter here](source_code/TaskManagerPart9.zip).

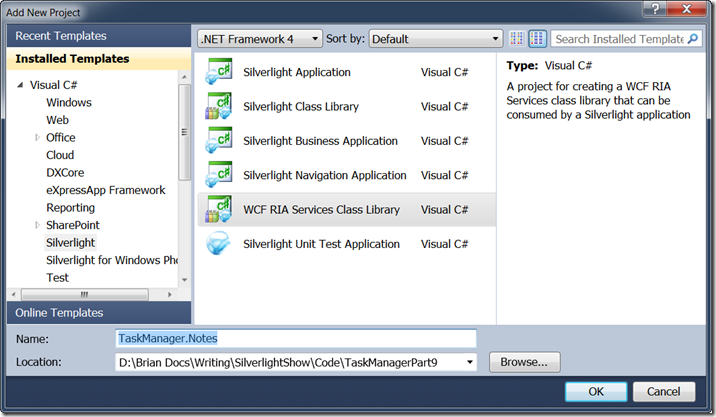
### Client-Server Project Relationships

One thing to understand up front is that there are some architectural constraints implied by the use of WCF RIA Services. Because of the way the RIA Services code generation process works, if you want to consume a WCF RIA DomainService from a client project, that client project has to have a link to the server project that contains the domain services you want to consume. That means that a single client project can only point to a single server project. You can have multiple client projects that point to the same server project, and the code generation will happen in each client project. But then if those projects are all used in the same scope, you will have duplicate types defined in the same scope, and will run into problems there. So to keep things clean, you will want to maintain a one-to-one correspondence between a single server project where a set of domain services is defined and a single client project where the client code for those services gets generated. However, the client project can just be a class library that can be reused across multiple modules or client applications. So your architecture will tend to look like this as you start to partition things:

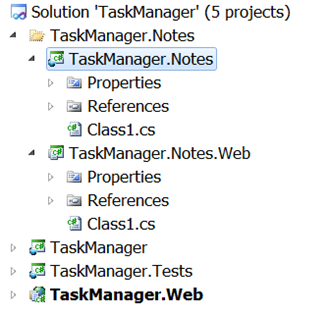
[](http://www.silverlightshow.net/Storage/Users/brian.noyes/Architecture_2.png)

### Adding a WCF RIA Services Class Library Project

The way to get started breaking things up is to add a new WCF RIA Services Class Library project to your solution. In this case, say I wanted to add some separable functionality to my application, such as the ability to add notes. I would select Add > New Project from Solution Explorer with the solution node selected, and pick the WCF RIA Services Class Library project type.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/AddRIALibrary_2.png)

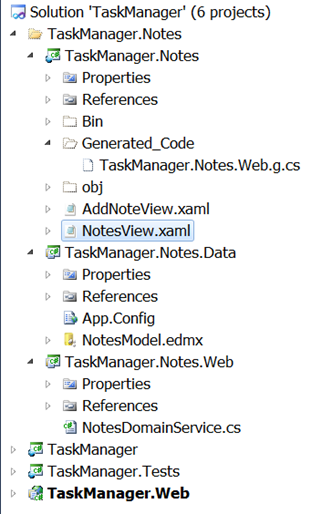
Once you add this project, you really are adding two projects, the client Silverlight Class Library and a server normal Class Library project. The RIA Services link will already be set between the projects, and it adds them to the solution under a new solution folder.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/ClassLibrariesInSolution_2.png)

You do not really have to use this project type to get this architecture though. You could just add a new Silverlight Class Library to the solution, then add a normal Class Library as well. Then you would just go to the project properties of the Silverlight Class Library and set the WCF RIA Services link drop down to point to the new server class library project. Then you would just have to add the WCF RIA Services references to the client and server projects. This just gets you all that done in one fell swoop. And since the client and server projects are linked, having them in a solution folder is a decent way to group them.

You would then add a new domain service to the TaskManager.Notes.Web project to get started defining your new service functionality and its associated entities. Its code generated client code will end up in the TaskManager.Notes client library. Then you would add a reference from the main client TaskManager Silverlight application to the TaskManager.Notes class library to start using those domain services in the main app. You could add new views and client functionality to that same class library as well. If you wanted to separate out your data access logic into its own class library, you would just add a new class library project and add your Entity Data Model into that project. Then add a reference to that class library from the class library that is going to contain your domain service. Remember to build before adding the new domain service so that it will see your entity framework model and let you pick from its entity types.

The resulting solution tree after adding some views and the data access library is shown below.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/SolutionTreeWithDomainService_2.png)

### Using Entities across Domain Services

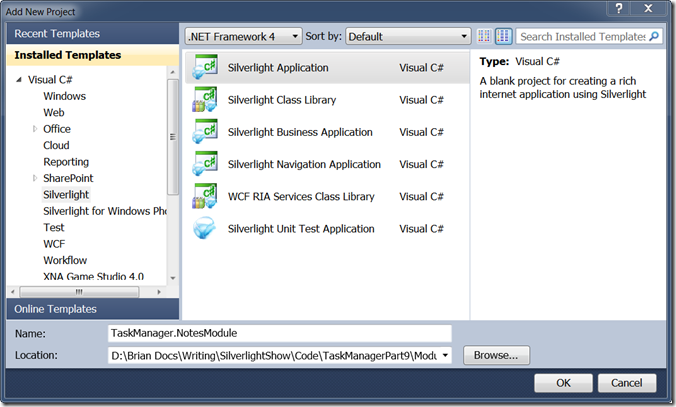
In WCF RIA Services version 1, there is a limitation that you cannot have two domain services being used by the same client application that expose the same entity type. This is just a limitation of the way the client side code generation is done, because it will try to generate the same entity type once for each service and you will have duplicate definitions. This is fixed in WCF RIA Services SP1, [which is available in beta form at the time of writing this here](http://www.silverlight.net/getstarted/riaservices/). But if you stick to the current release version, you will have to factor your vertical slices so that a given entity type is only used in one of the domain service vertical slices.

### Breaking Your Client Application into Multiple Modules

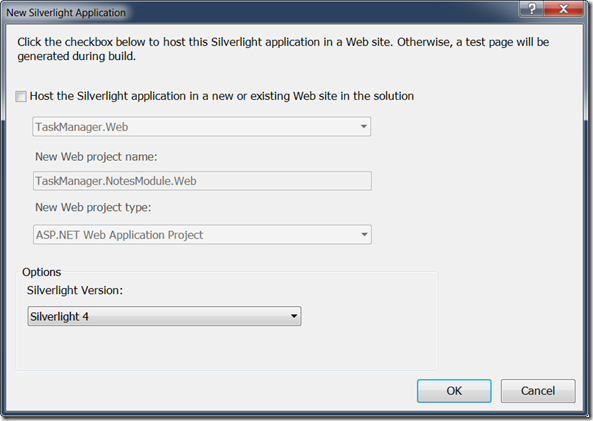
As mentioned earlier, using MEF or Prism you can break your client architecture into modules – or separate chunks of functionality that can be downloaded asynchronously at runtime. For this chapter, I’ll just use MEF directly. I’ll be doing some articles on Prism 4 in the near future as well.

To do this, you need to put your functionality in a project that compiles to a separate XAP file from the main application. That requires you to pick the Silverlight Application template when creating the project, as opposed to a Silverlight Class Library or a WCF RIA Services Class Library project.

If I were going to do something similar to what I showed earlier in the chapter but wanted that new notes functionality to be downloaded separately as a module the first time it is used, the first step would be to create a new Silverlight Application project in my solution named TaskManager.NotesModule.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/AddModuleProject_2.png)

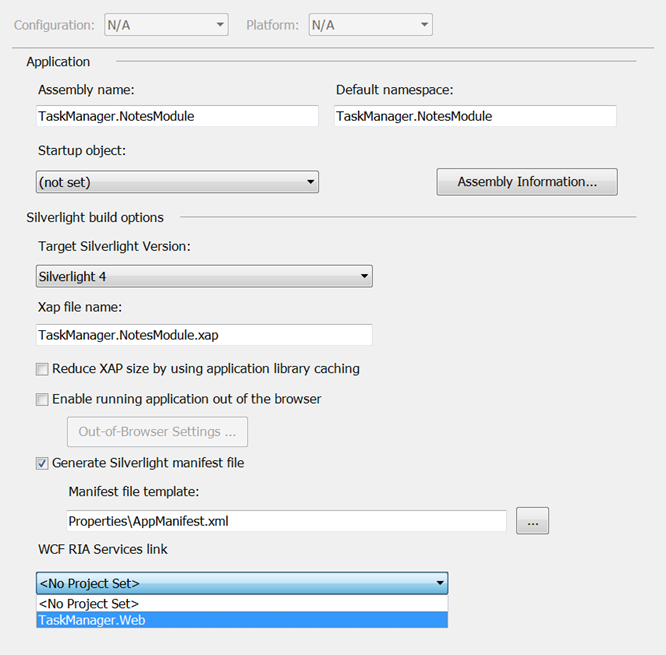
In the pop up that prompts for creating a server project, uncheck the box for hosting the application.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/NoServerProject_2.png)

After the project is created, delete App.xaml and MainPage.xaml from the new project.

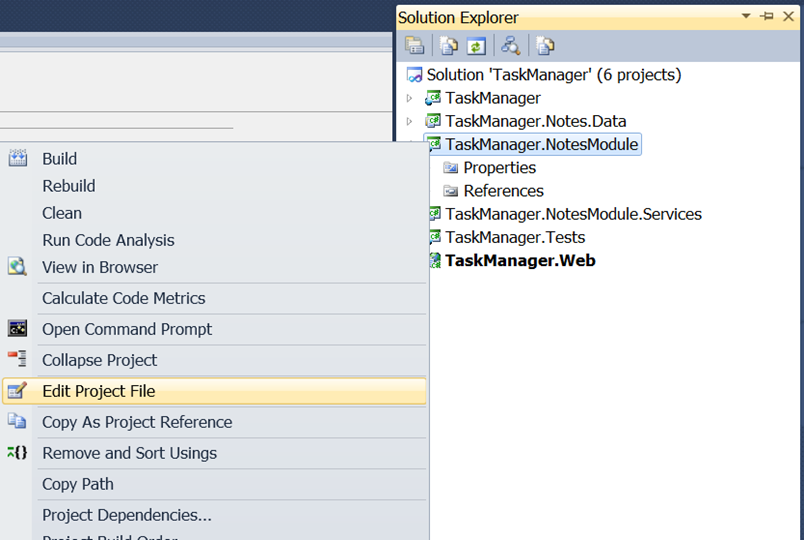
Next you will add the server class library where the domain service will live. Add a new Windows Class Library project and call it TaskManager.NotesModule.Services. You can delete Class1.cs from the new class library project as well. You could then add a domain service to that project, possibly using an entity data model defined in a separate class library as discussed earlier.

If you open the TaskManager.NotesModule Silverlight project settings at this point, the first thing you will need to do is set the startup object to (not set). That is because this project is not intended to be a standalone Silverlight application, but we are just using the Silverlight Application project type because its build output is to produce a XAP file with the contents of the project.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/ModuleProjectSettings_2.png)

The trick comes when you go to set up the RIA Services link.  If you drop down the WCF RIA Services link setting, you will not see the class library that contains your domain service listed as an option. This is really just a bug in the tooling. That drop down really just sets a relative path to the server project in the client project’s csproj file (or vbproj). Since the tool won’t let you set it correctly, you have to open the csproj file and edit it directly.

To do so, right click on the project and select Edit Project File.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/EditProjectFile_2.png)

If that option is not available for you, you can also just open the .csproj in any text editor. The setting you are looking for.is called LinkedServerProject. You need to fill it in with the relative path to the .csproj file of the server project. For example, in my case, the path is:

After adding this and saving the file and closing it, you can then right click on the project in Solution Explorer and select Reload Project. If you then go look at the project settings again, you will see it is now showing the right server project. If you build, you will get the client generated code in your module project from the linked server project.

### Using MEF To Dynamically Load the Module

Now all that is left is to load the module dynamically with MEF. I don’t have room for a full lesson on MEF, but what I am going to use here is MEF’s ability to download a separate XAP file asynchronously and then plug in the parts it finds in that XAP to the application dynamically. For a great overview of this capability, I recommend you check out [this Silverlight TV Episode with Glenn Block](http://johnpapa.net/silverlight/silverlight-tv-11-dynamically-loading-xaps-with-mef/).

The first step is to add the MEF System.ComponentModel.Composition reference to the TaskManager.NotesModule project. You can then mark the parts you want to plug in with appropriate **Export** attributes. In this case, I am going to plug in a single **ChildWindow** derived view called **NotesView**. So I add the following **Export** to the code behind of that view:

[Export(typeof(ChildWindow))]

public partial class NotesView : ChildWindow

{

}

Next, I need to add a little code the main app to download the separate XAP file asynchronously when appropriate. First step is to add references to the TaskManager project to System.ComponentModel.Composition.Initialization and System.ComponentModel.Composition. Then I add the following code to the code behind of the main page:

[Import]

public ChildWindow PlugInPopup { get; set; }

bool plugInsInitialized = false;

private void button1\_Click(object sender, RoutedEventArgs e)

{

if (!plugInsInitialized)

{

plugInsInitialized = true;

var deployment = new DeploymentCatalog("TaskManager.NotesModule.xap");

deployment.DownloadCompleted += (s, e2) =>

{

CompositionInitializer.SatisfyImports(this);

ShowPopup();

};

CompositionHost.Initialize(new DeploymentCatalog(), deployment);

deployment.DownloadAsync();

}

else

ShowPopup();

}

private void ShowPopup()

{

if (PlugInPopup != null)

PlugInPopup.Show();

}

The **PlugInPopup** property will be populated by MEF after the extra XAP is downloaded. Constructing a **DeploymentCatalog** with a relative path to the XAP file in the host site of this application allows it to download that XAP file asynchronously when told to do so with the **DownloadAsync** call. You can see that the code subscribes to the completed event for that download, and then calls **SatisfyImports** to get the container to do dependency injection on this already existing view. At that point the **PlugInPopup** property gets populated, and is then shown.

The last part to making this work is to add the TaskManager.NotesModule project to the host Web site (it does not need a test page) so that it is in the ClientBin directory and can be downloaded.

The key point here is the need to manually edit the project file to point to the right server project where your domain services live. Just because Visual Studio doesn’t always let you point to any project in your solution (or outside of the solution for that matter), the only requirement is that the LinkedServerProject have a relative path to a compiled project that contains domain services. So a simple edit of the project file gets you want you want to organize your projects however you want, as long as you maintain that one-to-one relationship between client and server projects.

### Summary

When it comes to organizing and structuring your solution, you can see that you have good flexibility to start breaking up chunks of functionality on the client and server sides into separate libraries and modules however it makes the most sense for your project. You should think in terms of breaking out vertical slices of functionality, composed of a server library project that contains a domain service or several that are related and all their supporting functionality and definitions on the server side. You will link that to a single client project, typically a Silverlight Class Library project or Silverlight Application project if you are trying to be more modular and download modules separately as separate XAPs. Even though you can have a single server project linked from multiple client projects, it will generally cause problems to do so within the same application because of duplicate definitions. So using client side class libraries that just contain the code generated RIA Services code and then referencing that class library from wherever that functionality is needed within the client application gives you flexibility to compose the client side however you want. Also remember that in WCF RIA Services SP1, the constraint on having one domain service “own” a single entity type is lifted.

[Sample code](source_code/TaskManagerPart9.zip)

# Chapter 10: Exposing Domain Services To Other Clients

### 

### Introduction

As mentioned in chapter 1 of this ebook, WCF RIA Services only supports code generating the client proxy code in Silverlight projects. However, that does not mean you cannot call your domain services from other clients. If I were not going to have a Silverlight client application as my primary client application, I would not bother defining my services as domain services. I would instead define normal WCF services or possibly WCF Data Services. To me, most of the benefit of WCF RIA Services is in the code generated client proxy code and client side framework. The validation support, the security model, the service error handling, and the deferred query execution are the things I think are most compelling about using RIA Services.

But If I do have a Silverlight client and use RIA Services, I probably don't want to have to implement a separate set of services for my non-Silverlight clients. The good news is, you don't have to. It is easy to expose additional endpoints from your domain services that can be consumed by other clients. In this chapter, I'll show you how to enable those endpoints, and will show what is involved in consuming your RIA domain services from non-Silverlight clients. Your options include exposing a read-only OData service endpoint, a full functionality SOAP endpoint compatible with the basicHttpBinding in WCF, or a full functionality REST JSON encoded endpoint.

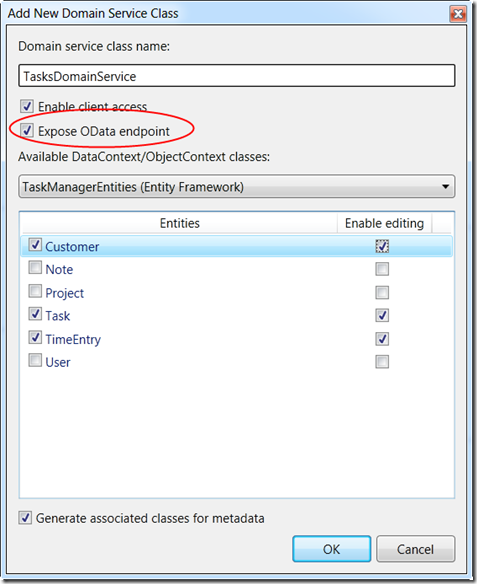
You can [get the source code for this chapter here](source_code/TaskManagerPart10.zip).

### Exposing an OData Endpoint From Your Domain Service

OData is short for the Open Data Protocol. It is a REST-based web service protocol for exposing data for querying via web services, and optionally allowing updates via that web service as well. You can read up on OData at <http://www.odata.org/>. OData uses the ATOM protocol for encoding the data in the HTTP body of the REST messages that flow from and to your service. OData allows you to express a complex query through parameters in the URL that is used to address the service. OData supports a subset of the common LINQ query operations such as filtering (the Where operation in LINQ), projection (the Select operation in LINQ), and paging (Take and Skip operations in LINQ). Additionally, the OData protocol allows you to send inserts, updates, and deletes for an exposed entity collection if the service allows it.

RIA Services allows you to expose a query-only OData endpoint (no updates) from your domain services. The exposed feed only allows you to retrieve the entire collection exposed by a query method. You cannot pass query filters or paging operations down to the service through the OData protocol, so the functionality is fairly limited at the current time. In a future release of WCF RIA Services they will probably support updates and more complex query operations, but for now you can basically just call your domain service query methods and return the collection that the server method returns without being able to filter from the client side.

This capability is part of the core WCF RIA Services libraries. All you need to do is remember to check the box when you first create your domain service as shown in the following figure.

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/Add%20Domain%20Service_2.png)

If you forgot to do that when you created the domain service, don’t fret. You can always add a new domain service to the project and check the box for that domain service, and then delete that domain service. Checking the box adds a **sectionGroup** to your config file, adds a new **domainServices** section to the **system.serviceModel** part of your config file, and adds another reference to the project to the System.ServiceModel.DomainServices.Hosting.OData.dll library. Additionally, the **[Query]** attribute is added to each of the entity query methods added by the wizard. If you forget to check the box, you will need to add those attributes yourself as well.

[Query(IsDefault=true)]

public IQueryable<Task> GetTasks()

{ ... }

[Query(IsDefault = true)]

public IQueryable<TimeEntry> GetTimeEntries()

{ ... }

[Query(IsDefault = true)]

public IQueryable<Customer> GetCustomers()

{ ... }

The sectionGroup it adds to the config file looks like this:

<system.serviceModel>

<domainServices>

<endpoints>

<add name="OData"

type="System.ServiceModel.DomainServices.Hosting.ODataEndpointFactory,

System.ServiceModel.DomainServices.Hosting.OData, Version=4.0.0.0,

Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

...

</endpoints>

</domainServices>

...

</system.serviceModel>

This endpoint does have metadata turned on, so clients can easily generate client proxy code from the endpoint like they would from any other WCF service. The address that this endpoint is exposed on is just the base domain service address (hosted server address + fully qualified domain service class name with dots replaced by dashes + .svc) with /odata appended to it.

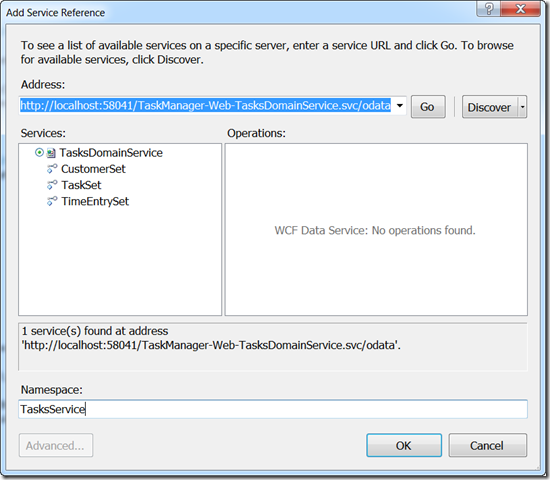
### Consuming the OData Endpoint From a .NET Client

What if you want to consume that OData feed? There are a variety of tools out there that can consume OData feeds, including the [OData Explorer](http://www.silverlight.net/content/samples/odataexplorer/), a plug in for Excel, and other tools. If you want to consume that data by querying it via services from another .NET client, it is very easy because Visual Studio can generate a client proxy for you in any .NET project. There is also a command line tool called [DataSvcUtil.exe](http://msdn.microsoft.com/en-us/library/ee383989.aspx) that can do the same client code generation, making it easy to consume the feed.

To demonstrate this, I can add a WPF Application project to the solution. I then select Add Service Reference from that project and enter the address to the odata feed:

http://localhost:11557/TaskManager-Web-TasksDomainService.svc/odata

You should see that the service is found, and you will see a collection set for each of your entities that you have the**[Query(IsDefault=true)]** attribute on:

[](http://www.silverlightshow.net/Storage/Users/brian.noyes/Add-Service-Ref-OData.png)

After you click OK, a client proxy will be generated. The generated OData proxy is quite different than a normal WCF service proxy. Instead of exposing methods that you call on the service, it exposes the entity sets. With normal OData services, you can form LINQ queries on those entity sets, and when you iterate over that expression (or call **ToList** or **Count** or other LINQ operators that do not defer execution), they will actually execute on the server side. The OData proxy sends the query expression tree to the server in a similar way to how WCF RIA Services does, by forming a query string from the expression tree of the LINQ expression as I explained in [Part 3](http://www.silverlightshow.net/items/WCF-RIA-Services-Part-3-Updating-Data.aspx).

Unfortunately, the RIA Services implementation only allows you to ask for the entire entity set via OData. If you try to use other LINQ expressions such as **Where** or **Take**, you will get an error back that says “Query options are not allowed.” However, you can use the OData feed to retrieve all the entities exposed by a domain service query method.

For example, I could execute the following code in the WPF client:

Uri serviceUri = new Uri("http://localhost:58041/TaskManager-Web-TasksDomainService.svc/odata");

TasksDomainService proxy = new TasksDomainService(serviceUri);

List<Task> tasksBefore2Jun = proxy.TaskSet.ToList();

This would return the whole list of **Tasks** from the server, and then the client could present that data. However, if it allowed the user to edit the data, there would be no way to send the changes back to the server via the OData endpoint. Notice that the way the proxy works is by exposing entity sets as a property on the proxy itself. Also notice that the proxy requires a URL to the service on construction. There is no contract or binding associated with an OData proxy and you pass the address in through the constructor, so there is no need for client configuration of the proxy either.

### Exposing a SOAP Endpoint From Your Domain Service

If you want to be able to execute your query and update methods from other clients, then you can use the SOAP or JSON endpoints that can also be enabled on your domain service. These require that you download and install the [RIA Services Toolkit](https://www.silverlight.net/getstarted/riaservices/) in addition to having the core RIA Services functionality that you get through the [Silverlight 4 Tools for Visual Studio 2010](https://www.silverlight.net/getstarted/riaservices/).

The SOAP endpoint is a WCF **basicHttpBinding** compatible endpoint that can be easily consumed by just about any platform that speaks SOAP. To add the SOAP endpoint, you just add another endpoint in the domainServices section in your config, in the same place as the OData endpoint shown earlier. It looks like this:

<system.serviceModel>

<domainServices>

<endpoints>

<add name="soap"

type="Microsoft.ServiceModel.DomainServices.Hosting.SoapXmlEndpointFactory,

Microsoft.ServiceModel.DomainServices.Hosting, Version=4.0.0.0,

Culture=neutral, PublicKeyToken=31bf3856ad364e35" />

...

</endpoints>

</domainServices>

...

</system.serviceModel>

You will also need to add a reference in the web host to Microsoft.ServiceModel.DomainServices.Host, which is where the **SoapXmlEndpointFactory** type is defined as you can see from the config code above.

That endpoint does have metadata turned on, so clients can easily generate client proxy code from the endpoint like they would from any other WCF service. The address that this endpoint is exposed on is just the base domain service address with /soap appended to it.

### Consuming the SOAP Endpoint From a .NET Client

To consume the SOAP Endpoint, you just do a normal Add Service Reference in the client project, or use svcutil.exe, or hand-code a proxy class using the **ClientBase<T>** base class. Using Add Service Reference is the easiest if you are new to WCF Services.

To add a service reference to the SOAP endpoint, just point Add Service Reference or svcutil.exe to the default address of your domain service, http://localhost:58041/TaskManager-Web-TasksDomainService.svc for the sample application. That will generate the compatible proxy and configuration code for the client.

Then you could write client code like the following to retrieve the Tasks collection and make an update to one of the tasks and send it back to the service to persist the change:

TasksDomainServicesoapClient proxy = new TasksDomainServicesoapClient();

// Retrieve the full collection,

// no ability to filter server side unless additional methods exposed

QueryResultOfTask result = proxy.GetTasks();

Task[] tasks = result.RootResults; // Extract the real collection from the wrapper

// Make a modification

tasks[0].TaskName = "Modified by SOAP Client";

// Wrap it in ChangeSetEntry

ChangeSetEntry changeEntry = new ChangeSetEntry();

changeEntry.Entity = tasks[0];

changeEntry.Operation = DomainOperation.Update;

// Send the changes back to the server as an array of ChangeSetEntries

proxy.SubmitChanges(new ChangeSetEntry[] { changeEntry });

Task[] newFetchTasks = proxy.GetTasks().RootResults;

proxy.Close();

Most of the complexity in dealing with the SOAP endpoint is in wrapping up changes in the **ChangeSetEntries**. That type supports sending the original entity and the modified entry back as well for optimistic concurrency checking or so that the server side can optimize the query by knowing which properties have actually changed on the object. Other than the wrapping of the entities, this is just normal WCF proxy-based service calls.

In the sample code for this chapter, I turned off security to keep things focused on the basic mechanisms of exposing the endpoints and calling them. To secure the endpoints, you would again just leave the**[RequiresAuthentication]** attribute on the domain service and add an **AuthenticationDomainService** as discussed in chapter 7. On the client side, you would need to make a call to the authentication domain service first to establish a login session. You would also need to enable a cookie container on the proxy for both the authentication domain service endpoint and the other domain services you want to call. Finally, you would need to copy the cookie container from the authentication service proxy to the other proxies after logging in. For a great walkthrough on this in the context of a Windows Phone 7 client, see [this blog post by Marcel de Vries](http://blogs.infosupport.com/blogs/marcelv/archive/2010/10/08/using-authenticated-ria-services-on-your-wp7-phone.aspx).

### Exposing a REST/JSON Endpoint

Exposing a REST/JSON style endpoint from your domain service that functions just like the SOAP one just described, it is just another endpoint declaration in your configuration file.

<add name="JSON"

type="Microsoft.ServiceModel.DomainServices.Hosting.JsonEndpointFactory,

Microsoft.ServiceModel.DomainServices.Hosting, Version=4.0.0.0,

Culture=neutral, PublicKeyToken=31bf3856ad364e35"

/>

You can then use your favorite approach such as a **WebClient** or **HttpWebRequest** in .NET to issue the HTTP request to the REST endpoint, and can use something like the WCF **DataContractJsonSerializer** to decode and encode the JSON payload in the HTTP body of the message. The address scheme is based on the addressing scheme of WCF service methods that you expose via REST. For example, to call the **GetTasks** method, you would just address http://localhost:58041/TaskManager-Web-TasksDomainService.svc/json/GetTasks.

### Summary

As you can see, it is a fairly simple matter to expose the additional endpoints for OData, SOAP, and REST/JSON from your domain services. Because of the limitations on the OData endpoint in the current release, I find that one to be the least useful. However, the SOAP and REST endpoints do make it fairly easy to consume your domain services from other platforms. If I needed to provide CRUD services for a set of entities and needed to write client applications on multiple platforms with the full set of functionality, and wanted to make it as easy as possible for others to write clients for my services, I would not use WCF Data Services for that. I would use either WCF Data Services to expose a fully functional OData endpoint, or I would write normal WCF Services where I was not constrained by the server side model of WCF Data Services. However, if I was writing a complex Silverlight application that was the primary client application, and just wanted to be able to expose some of the same entity CRUD functionality to other clients without needing to write separate services for them or give up the client side benefits of WCF RIA Services for my Silverlight client, then these additional endpoints are just what is needed.

So that brings me to the end of this ebook on WCF Data Services. Keep an eye on my blog at <http://briannoyes.net/> for additional posts about WCF RIA Services, and I will probably write some other articles on this and other topics on [SilverlightShow](http://www.silverlightshow.net) as well. Thanks for reading, and please let me know any feedback you have.

[Source code](source_code/TaskManagerPart10.zip)

# About the Author

Brian Noyes is Chief Architect of [IDesign](http://www.idesign.net/), a Microsoft Regional Director, and Connected System MVP. He is a frequent top rated speaker at conferences worldwide including Microsoft TechEd, DevConnections, DevTeach, and others. He is the author of [Developing Applications with Windows Workflow Foundation](http://www.amazon.com/Developing-Applications-Workflow-Foundation-Training/dp/0321503139/ref=sr_1_1?ie=UTF8&s=books&qid=1289774458&sr=1-1), [Smart Client Deployment with ClickOnce](http://www.amazon.com/Smart-Client-Deployment-ClickOnce-Applications/dp/0321197690), and [Data Binding in Windows Forms 2.0](http://www.amazon.com/Data-Binding-Windows-Forms-2-0/dp/032126892X). Brian got started programming as a hobby while flying [F-14 Tomcats](http://en.wikipedia.org/wiki/Grumman_F-14_Tomcat) in the U.S. Navy, later turning his passion for code into his current career.

You can contact Brian through his blog at <http://briannoyes.net/> or on twitter @briannoyes.