

**dunn&churchill**

---

Help and Support

***Getting Started with Diamond Binding***

**Matthew Dunn**

## 1. Getting Started with Diamond Binding

---

### 1.1. Introduction

We're going to demonstrate Diamond Binding by building up a small example web application – one to catalogue cooking recipes. The application will be called *RecipeDemo*, and it will store cooking recipe instructions, ingredients and a list of units in a SQL Server database.

We're going to use an iterative approach to develop the *RecipeDemo*, as we want to show how Diamond Binding works well with an agile development process.

After following along with all iterations, you should be ready to create your own applications with Diamond Binding.

### 1.2. Creating the Database

First we need to create the demonstration database:

```
CREATE TABLE Recipe (  
  [Id] int IDENTITY(1,1) NOT NULL,  
  [Name] nvarchar(50) NOT NULL,  
  [Instructions] nvarchar(max) NOT NULL,  
  CONSTRAINT [PK_Recipe] PRIMARY KEY  
  ( [Id] ASC )  
 ) ON [PRIMARY]
```

Our demonstration database has a name, a place for instructions, and a simple auto-incrementing primary key.

### 1.3. Creating the Business Layer

In Visual Studio, create a new class library project with name *DiamondGettingStarted.Data*. Select the empty project in the *Solution Explorer* window and then open the *Project* menu. You should see a new *Diamond Binding* menu option, if there's not, ensure Diamond Binding has been installed and enabled in the *Add-In Manager* (in the *Tools* menu).

#### 1.3.1. Connect to Database

Select your project in the solution explorer. Underneath the *Diamond Binding* menu, select *Edit Current*. Use this dialog to connect to our recipe database by selecting an appropriate OLE DB provider, such as *Microsoft OLE DB Driver for SQL Server*.

If you are supplying a username and password, you will need to ensure you check the "Allow Password Saving" box so Diamond Binding can remember the password when it needs to resynchronise definitions.

#### 1.3.2. Configure Definitions

Once you have connected to the *RecipeDemo* database, the Diamond Binding project configuration form will be displayed. Leave the *General* tab settings at their defaults and click the *Definitions* tab. Tick the checkbox next to the *Recipe* table to indicate you wish to have definition files created for it. Press *Ok* and after a short while a new *Recipe* class will be added to the project containing all the fields in the *Recipe* table.

## 2. Creating the Application Project

---

Next we'll use our new business layer to demonstrate loading and saving recipe records, we'll just use a console application for this purpose.

Add a new console project to the solution, and add a reference to our business project. At this stage, you should also add a reference to the following *Diamond Binding* runtime assemblies;

- DunnChurchill.Data.DiamondBinding
- DunnChurchill.Data.DiamondBinding.ActiveRecord
- DunnChurchill.Data.DiamondBinding.Core
- NHibernate

These assemblies are located in the *C:\Program Files\Dunn & Churchill\Diamond Binding\Runtime* folder by default.

## 2.1. Configuring the Runtime

The simplest way to configure the Diamond Binding runtime is with a standard *App.config* file, add one of these to the console project and paste in the following xml:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
    <section name="activerecord"
type="DunnChurchill.Data.DiamondBinding.ActiveRecord.Framework.Config.ActiveRecord
SectionHandler, DunnChurchill.Data.DiamondBinding.ActiveRecord, Version=1.3.0.0,
Culture=Neutral, PublicKeyToken=6e7b61d5b7adddc4" />
  </configSections>
  <connectionStrings>
    <add name="SqlServer" connectionString="{your connection string}"/>
  </connectionStrings>
  <activerecord
namingstrategytype="DunnChurchill.Data.DiamondBinding.SqlServerNamingStrategy,
DunnChurchill.Data.DiamondBinding, Version=1.3.0.0, Culture=Neutral,
PublicKeyToken=6e7b61d5b7adddc4">
    <config>
      <add key="hibernate.connection.driver_class"
value="NHibernate.Driver.SqlClientDriver" />
      <add key="hibernate.dialect" value="NHibernate.Dialect.MsSql2000Dialect" />
      <add key="hibernate.connection.provider"
value="NHibernate.Connection.DriverConnectionProvider" />
      <add key="hibernate.connection.connection_string_name" value="SqlServer" />
    </config>
  </activerecord>
</configuration>
```

Replace *connectionString* with a valid *SQL Native Client* connection string to your database. This will likely be the same as the connection string created for you in the *Diamond Binding Project Configuration*, without the *Provider* item. Check that the version numbers for the *ActiveRecordSectionHandler* and the *SqlServerNamingStrategy* are set to the current version of *Diamond Binding* you have installed. (If you are pasting from an .xps document be careful of spaces being inserted in the *ActiveRecordSectionHandler* name).

One final step before we are ready to start persisting business objects, is to initialise the *Diamond Binding Runtime*. This can be done simply by pasting the following line of code at the start of your application (the using directives are shown here for clarity):

```
using DunnChurchill.Data.DiamondBinding.ActiveRecord;
using DunnChurchill.Data.DiamondBinding.ActiveRecord.Framework.Config;
...

// Initialize the Diamond Binding Runtime
ActiveRecordStarter.Initialize(typeof(Recipe).Assembly,
ActiveRecordSectionHandler.Instance);
```

## 2.2. Loading and Saving Objects

Now for the easy part, the base *ActiveRecord* class provides overloads for saving and loading records; the following code demonstrates how to persist a new *Recipe* record:

### 2.2.1. Save

The *Save* function persists any modifications to an existing record. If the record's primary key is also an *Identity* column, as is the case with our recipe database, *Save* is smart enough to execute an insert if the record is new, or executes an update if the record already exists.

```
//Create recipe and persist it to the database
Recipe recipe = new Recipe();
recipe.Name = "Honey Peanut Granola";
recipe.Instructions = "Test";
recipe.Save();
```

### 2.2.2. Create

The *Create* function inserts a new record into the database. If the primary key is not an *Identity* column this function should be called instead of *Save* to indicate the record should be inserted.

```
//Create recipe and persist it to the database
Recipe recipe = new Recipe();
recipe.Name = "Honey Peanut Granola";
recipe.Instructions = "Test";
recipe.Create();
```

### 2.2.3. Find

The *Find* function loads a record by its primary key.

```
Recipe recipe = Recipe.Find(1);
Console.WriteLine("Recipe: {0}", recipe.Name);
```

### 2.2.4. Find All

The *FindAll* function returns an array of every record in the table.

```
Recipe[] recipes = Recipe.FindAll<Recipe>();
foreach (Recipe recipe in recipes)
{
    Console.WriteLine("Found recipe '{0}'", recipe.Name);
}
```

### 2.2.5. Find By Property

The *FindByProperty* function returns an array matching a property.

Go to the *Recipe.cs* class, and add the following function:

```
public static Recipe[] FindByName(string name)
{
    return Recipe.FindAllByProperty("Name", name);
}
```

Using this function is simple:

```
//Find a specific recipe by name
foreach (Recipe recipe in Recipe.FindByName("Honey Peanut Granola"))
{
    Console.WriteLine("Found recipe '{0}'", recipe.Name);
}
```

Note. Diamond Binding is smart enough not to overwrite the *Recipe.cs* file, so it's appropriate to place your own business logic there.

Remember, when dealing with the *Find* series of functions, you are querying the object model, so field and class names refer to the names as they appear in code.

### 2.3. Summary

In this section we've examined how to use Diamond Binding to map a business object to an SQL table, how to persist a record, and how to add a business logic query to one of the Diamond Binding managed classes.

## 3. Extending the *RecipeDemo* Application

Our original example only contained one table. To demonstrate more features of Diamond Binding, we'll start with our recipe database and extend it with new tables.

### 3.1. Creating the Database

First we need to create the demonstration database:

Table *Recipe* contains recipe name and instructions.

```
CREATE TABLE Recipe(  
  [Id] int IDENTITY(1,1) NOT NULL,  
  [Name] nvarchar(50) NOT NULL,  
  [Instructions] nvarchar(max) NOT NULL,  
  CONSTRAINT [PK Recipe] PRIMARY KEY  
  ( [Id] ASC )  
 ) ON [PRIMARY]
```

The *Unit* table contains a list of units, e.g. 'teaspoons', 'cups' and 'grams'.

```
CREATE TABLE [dbo].[Unit](  
  [Id] [int] IDENTITY(1,1) NOT NULL,  
  [Name] [nvarchar](50)  
  CONSTRAINT [PK_Unit] PRIMARY KEY CLUSTERED ([Id] ASC)  
 )
```

Table *Ingredient* contains ingredients for a recipe, in addition to a unit and quantity. For example representing '3 teaspoons of sugar'.

```
CREATE TABLE [dbo].[Ingredient](  
  [Id] [int] IDENTITY(1,1) NOT NULL,  
  [Name] [nvarchar](50) NOT NULL,  
  [Quantity] [int] NOT NULL,  
  [Unit] [int] NOT NULL REFERENCES [dbo].[Unit] ([Id]),  
  [Recipe] [int] NOT NULL REFERENCES [dbo].[Recipe] ([Id]),  
  CONSTRAINT [PK_Ingredient] PRIMARY KEY CLUSTERED ([Id] ASC)  
 )
```

### 3.2. Creating the Business Layer

Once you've finished creating the new database tables. Load the *Diamond Binding Recipe* solution you created in the last tutorial. We're going to tell *Diamond Binding* about the new tables and relationships, so go to the *Diamond Binding* menu and select *Edit Current* then click the *Definitions* tab again.

First ensure all three tables are selected (*Unit*, *Recipe*, and *Ingredient*) *Diamond Binding* will automatically enable all sensible relationships by default.

In this particular scenario we aren't interested in the list of *Ingredients* which happen to reference a particular *Unit*, because the *Unit* table is just a pre-populated list (such as 'teaspoon'.) So remove this relationship by un-checking the *Ingredient.Unit* relationship under the *Unit* table. All the other settings are fine, so just go ahead and press *Ok*.

*Diamond Binding* will now automatically resynchronise the project definitions, and you should see the new classes appear.

### 3.3. Customising the Business Layer

When we designed the database, we added a foreign key to the *Ingredient* table, pointing to the *Recipe* table, because we intended to allow *Recipe* to have a list of *Ingredients*. In the *Recipe* class *Diamond Binding* will have created a property *Ingredient\_Recipe*; a strongly typed list of *Ingredients*. This property is named after the *Ingredient* table, and that table's foreign key column, *Recipe*.

We'll rename this property to *Ingredients*, because it's more convenient.

Open the user class *Recipe.cs* and add the following *Ingredients* property to the class.

```
public IList<Ingredient> Ingredients
{
    get
    {
        return Ingredient_Recipe;
    }
}
```

We've successfully renamed the property *Ingredient\_Recipe* to *Ingredients* by wrapping the *Ingredient\_Recipe* property generated by *Diamond Binding*. However, the old *Ingredient\_Recipe* property is still publically accessible. To change this, in the *Diamond Binding Project Configuration* edit the definition for the *Ingredient.Unit* relationship and change the field accessibility from *Default* to *Protected*.

Note, if you rename a property this way, you still have to refer the property by its original name when using *Hibernate Query Language*.

## 3.4. Loading and Saving Objects

### 3.4.1. Saving a New Recipe

Add the following code to the console application:

```
//Create a new Recipe - Chicken Fondue
Recipe recipe = new Recipe();
recipe.Name = "Honey Garlic Chicken Fondue";
recipe.Instructions = "Empty";

//Create an new Ingredient - Chicken
Ingredient i = new Ingredient();
i.Name = "Chicken";
i.Quantity = 600;

//Create a new unit - grams
Unit unit = new Unit();
unit.Name = "grams";

i.Unit = unit;

recipe.Ingredients.Add(i);
//Persist to database.
recipe.Save();
```

Here we've saved a Chicken Fondue recipe in the database containing one ingredient ; 600 grams of chicken. All we had to do was set some properties, add a new ingredient to the *Ingredients* collection and finally call *recipe.Save()*.

We only had to call save on the *Recipe* class because Diamond Binding knows about the *Ingredient.Recipe* foreign key relationship, and the cascade setting is set to *SaveUpdate*.

## 3.5. Summary

In this section we've seen how Diamond Binding makes the relationship between our business objects very intuitive. Hopefully by now you've got an idea of some of the power Diamond Binding has to offer. In future guides we'll examine more complex features, such as one-to-many relationships, inheritance, expressions, and cascading updates.