Building an MVC 3 App with Code First and Entity Framework 4.1

Julie Lerman

March 2011

Watch a video of this content

Download the code for this article:

- C# version (VS2010)
- VB version (VS2010)

Microsoft's ADO.NET Entity Framework (EF) simplifies data access by allowing you to avoid working directly with the database in your code. Instead you can retrieve data by writing queries against strongly typed classes letting the Entity Framework handle the database interaction on your behalf. EF can also persist changes back to the database for you. In addition to this benefit, you will also benefit from the EF's comprehension of relationships. This means you will not be required to write extra code to specify joins between entities when expressing queries or simply working with your objects in memory.

EF provides you with three ways to describe the model of your entities. You can begin with a legacy database to create a model. You can design a model in a designer. Or you can simply define classes and let EF work with those. This last tactic is referred to as *code first*, because the first thing you do is write code to describe your entities.

In this whitepaper, I will walk you through creating a simple MVC 3 application using Entity Framework's code first technology to describe your classes and manage all of your data access.

Overview

In this walkthrough you will build pieces of a blogging application. The walkthrough will not result in a fully functional blogging application, but instead it will use the blog classes to demonstrate code first's features. You will:

- Define a set of classes to represent the entities for a blog — Blog, Post and Comment.
- Reference the Entity Framework code first assembly.
- Create a DbContext to manage the Blog classes and data access.
- Build a simple ASP.NET MVC 3 application that will let you view, add, edit and delete blogs.

Creating the MVC Application

For this demo, all of the code will live inside of an MVC 3 project so the first step will be to create that project. You'll use a template that will set up much of the structure for the application.

If you have not installed MVC 3 you can find the download and instructions at http://www.asp.net/mvc/mvc3.

1. In Visual Studio 2010, add a new project by selecting the File menu, then New and then Project.
2. In the Search Installed Templates text box (top right), type MVC 3 to filter down the list of project templates.
3. Select ASP.NET MVC 3 Web Application using your language of choice. This walkthrough will use C# code, but you can find both C# and Visual Basic versions in the sample solutions download.
4. Name the project MVC3AppCodeFirst and then click OK.
5. In the New ASP.NET MVC 3 Project wizard, choose Internet Application.
6. Leave the default values of the other options and click OK.

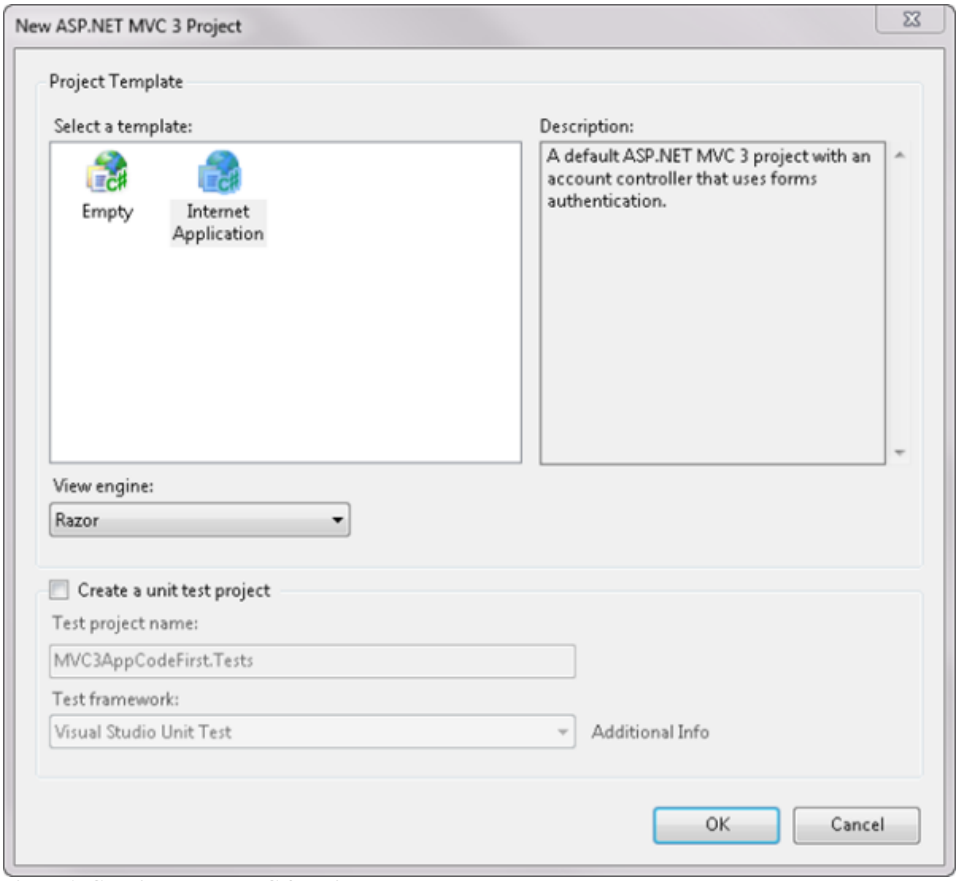

Figure 1: Creating a new MVC 3 Project

Visual Studio will create a full project structure for you including folders that contain pre-built controller classes, models and views. In fact you can run the application already to see it's default content which is simply a message welcoming you to MVC.

Creating the Model Classes for the Blogging Application

The M in MVC stands for Model and represents your domain classes. Each class is thought of as a model. You'll store your model classes — Blog, Post and Comment — in the Models folder. To keep things simple for your first look at code first, we'll build all three classes in a single code file.

1. In Solution Explorer, right click the Models folder.
2. Select Add from the menu and then from the bottom of its context menu choose Class.
3. In the Add New Item dialog, change the new class name to BlogClasses.
4. A new file, BlogClasses will be created.
5. Remove the BlogClasses class from within the file and replace it with the three classes listed below.

```
public class Blog
{
    public int Id { get; set; }
    public string Title { get; set; }
    public string BloggerName { get; set;}
    public virtual ICollection<Post> Posts { get; set; }
  }

public class Post
{
    public int Id { get; set; }
    public string Title { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int BlogId { get; set; }
    public ICollection<Comment> Comments { get; set; }
}

public class Comment
{
    public int Id { get; set; }
    public DateTime DateCreated { get; set; }
    public string Content { get; set; }
    public int PostId { get; set; }
    public Post Post { get; set; }
}
```

Creating an Entity Framework Context to Manage the Classes and Database Interaction

Now that you have classes, you'll need to bring in the Entity Framework to manage those classes while it provides data access and change tracking for their object instances. Entity Framework's DbContext class performs this job. We'll create a class that inherits from DbContext and knows how to serve up and manage the Blog, Post and Comment objects. The Entity Framework will take care of bridging the classes and a database.

Before you can use the DbContext, you'll need to create a reference to the Entity Framework 4.1 API. When you installed MVC 3, it added a new feature to Visual Studio 2010 called Nuget. Nuget allows you to easily find and install reference assemblies from the internet.

1. Select the MVC3AppCodeFirst project in Solution Explorer.
2. From the Tools Menu, choose Library Package Manager which has a sub-menu.
3. From the sub-menu choose Package Manager Console.
4. At the console's PM prompt type *install-package EntityFramework* then hit enter.

When the package is installed, you should see the "success message" shown in Figure 2.

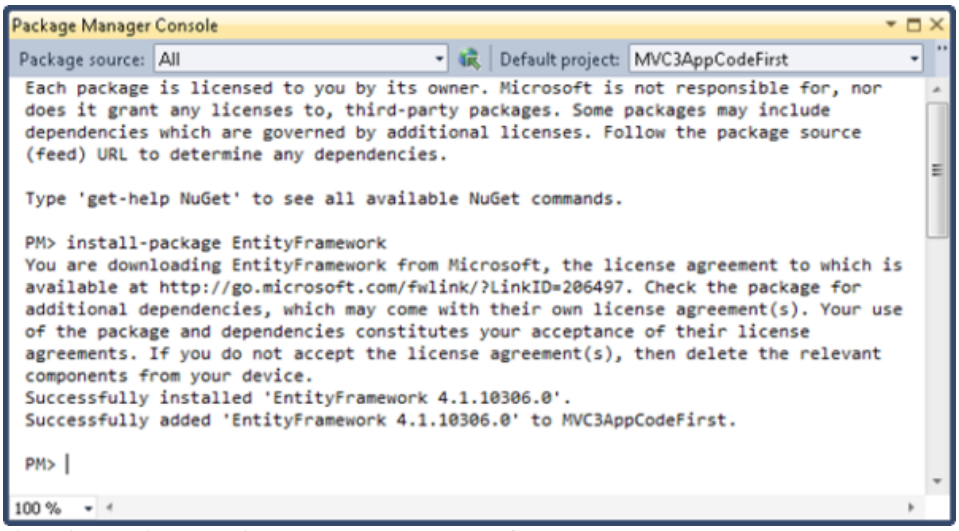

Figure 2: Installing the Entity Framework 4.1 package reference

Also there will be a new reference listed in the project references called EntityFramework. This is the assembly that contains the Code First runtime.

1. Add a new class to the Models folder and name it BlogContext.
2. Modify the class to match the following code:

```
using System.Data.Entity;
namespace MVC3AppCodeFirst.Models
{
  public class BlogContext : DbContext
  {
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<Comment> Comments { get; set; }
  }
}
```

These DbSet properties of the BlogContext are the key Entity Framework's ability to execute database queries on your behalf. As an example, BlogContext.Blogs will execute a query of the Blogs table in the database and return a set of Blog object instances based on the Blog class you defined earlier.

But where is this database? Code first depends on a default presumption about the location of a database. If you don't specify otherwise, EF will look for the database in the default location and if it doesn't exist, it will create it for you. Also by default, this database will be a SQL Express database with the name derived from the strongly typed name of the context and its file will be in the SQL Express default data folder. For example:

```
C:\Program Files\Microsoft SQL Server\MSSQL10.SQLEXPRESS\MSSQL\

     DATA\MVC3AppCodeFirst.Models.BlogContext.mdf
```

We'll rely on this default behavior for this walkthrough.

Creating the Controller Logic for the Blog Class

The "C" in MVC stands for controller. A controller gathers up data and sends it to a view and responds to requests from the view. We'll create a controller that uses the BlogContext to retrieve data for us. In a more advanced application, you should separate logic further and would not be working with the BlogContext directly from the controller.

For the Blog class, we'll be creating one view that displays a list of available Blogs and another that lets users create new Blogs. Therefore the controller will need one method to gather up a list of blogs to present in the first view, another method to provide a view to enter information for a new blog and finally a method to handle saving that new blog back to a database.

The project template created a controller called Home with a set of views and another controller called Account with its own set of views. We'll ignore those and create our own controllers and views, starting with a controller for the Blog class.

1. Build the project by choosing Build from the Visual Studio menu and then Build MVC3AppCodeFirst from its drop-down menu.
2. In the Solution Explorer, right click the Controllers folder.
3. Click Add from its context menu and then Controller.
4. In the Add Controller window, change the Controller Name to BlogController and check the Add action methods checkbox.
5. Click the Add button.

The BlogController class will be created with ActionResult methods: Index, Details, Create, Edit and Delete.

The job of the Index method will be to return a list of Blogs to be displayed in a view. The default code returns an empty View. Change the Index method code to the following:

```
public ActionResult Index()
{
```

```
    using (var db = new BlogContext())
    {
        return View(db.Blogs.ToList());
    }
}
```

The revised method instantiates a new BlogContext and uses that to query for Blogs then return the results in a List. Entity Framework will take care of converting this into a SQL query, executing the query in the database, capturing the database results and transforming them into Blog objects. That single line of code, context.Blogs.ToList(), is responsible for all of those tasks.

Since you don't yet have a database, code first will create it the first time you run this application and execute the Index method. After that, code first will automatically use the database that is created. As stated earlier, we'll rely on the default behavior and won't need to be concerned any more about the database.

Creating the Index View to Display the Blog List

We'll add an Index view so that you can see what happens to the results of the Index method.

1. Right click anywhere in the Index method declaration (public ActionResult Index()).
2. In the context menu that opens click the Add View option.

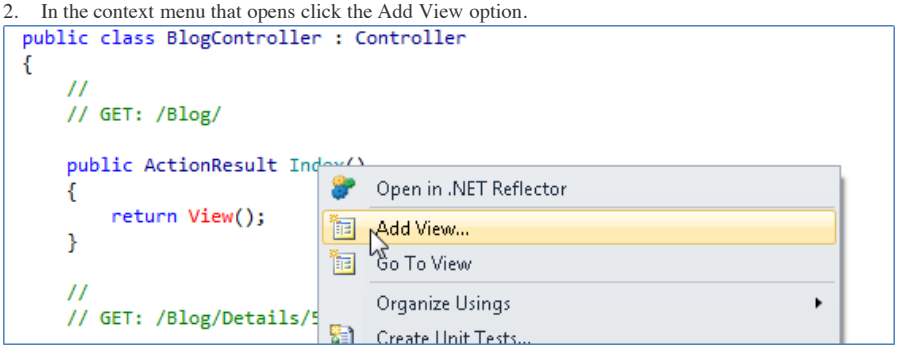

Figure 3: Adding a view to a controller ActionResult

3. The AddView dialog will open with the View name already set to Index.
4. Check the Create a strongly typed view checkbox.
5. Drop down the Model class option and select Blog.
6. From the Scaffold template options select List.

These selections will force Visual Studio to create a new web page with markup to display a list of Blogs. The markup for this web page will use the new MVC Razor syntax. Figure 4 shows the Add View dialog.

7. Click Add to complete the view creation.

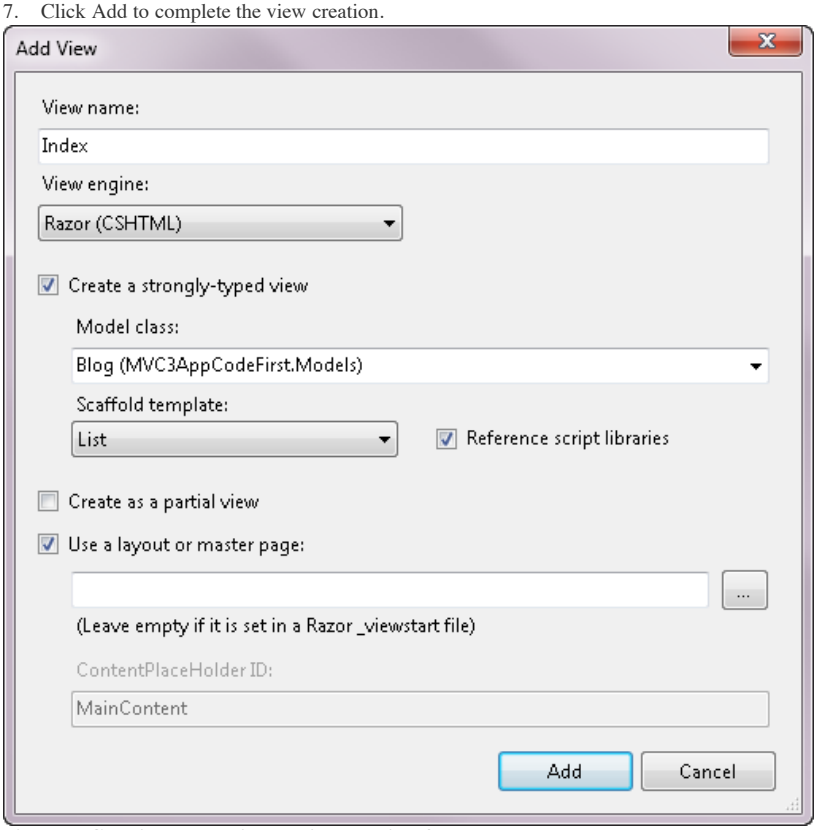

Figure 4: Creating a new view to display a list of Blog types

Visual Studio will create a Blog folder inside of the Views folder and add the new Index view to that folder as shown in Figure 5.

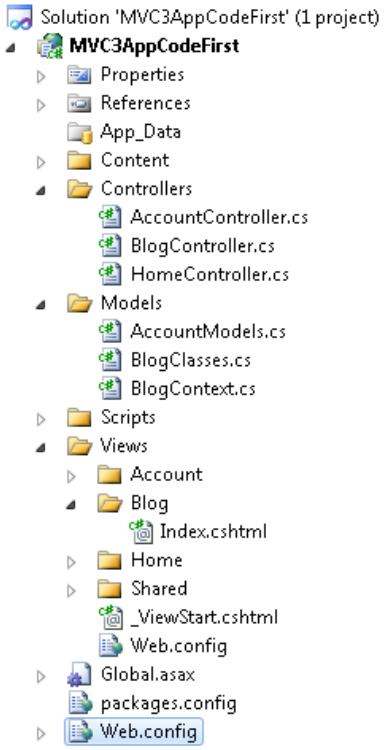

Figure 5: The new Blog Index View is added to the Views/Blog folder

Since the very first call to this method will create a new database, the list of blogs will be empty and you'll have to begin by creating new blogs. This is where the two Create methods come into play. Let's add in the Create logic and View before running the app. Note that you'll need to make a change to the global.asax file before running the app anyway. You'll do this shortly.

Adding Controller Logic to Create New Blogs

The first Create method does not need to return anything along with the View. Its View will be used to enter details for a new blog, so those details will start out empty. You don't need to make any changes to this method.

```
public ActionResult Create()
{
     return View();
}
```

The second Create method is of more interest. This is the one that will be called when the Create view posts back. By default, the template told this Create method to expect a FormCollection to be passed in as a parameter.

```
[HttpPost]
public ActionResult Create(FormCollection collection)
```

Because the Create view was created as a strongly-typed view, it will be able to pass back a strongly typed Blog object. This means you can change that signature so that the method receives a Blog type. That will simplify the task of adding the new blog to the BlogContext and saving it back to the database using Entity Framework code.

Modify the Create method overload (the second Create method) to match the following code:

```
[HttpPost]
public ActionResult Create(Blog newBlog)
{
     try
     {
         using (var db = new BlogContext())
         {
              db.Blogs.Add(newBlog);
              db.SaveChanges();
         }
         return RedirectToAction("Index");
      }
      catch
      {
        return View();
      }
}
```

In this method, you are taking the new blog that is passed in and adding it to the BlogContext's collection of Blogs, then calling Entity Framework's SaveChanges method. When .NET executes SaveChanges, it will construct and then execute a SQL INSERT command using the values of the Blog property to create a new row in the Blogs table of the database. If that is successful, then MVC will call the Index method which will return the user to the Index view, listing all of the existing blogs including the newly created blog.

Adding a View for the Create Action

Next, create a view for the Create ActionResult as you did for Index. You can right click either of the Create methods but you only need to build one Create view. Figure 6 shows the settings for the Create view. Notice that the scaffold template is Create. That will result in a very different looking web page than the List you chose for the Index view.

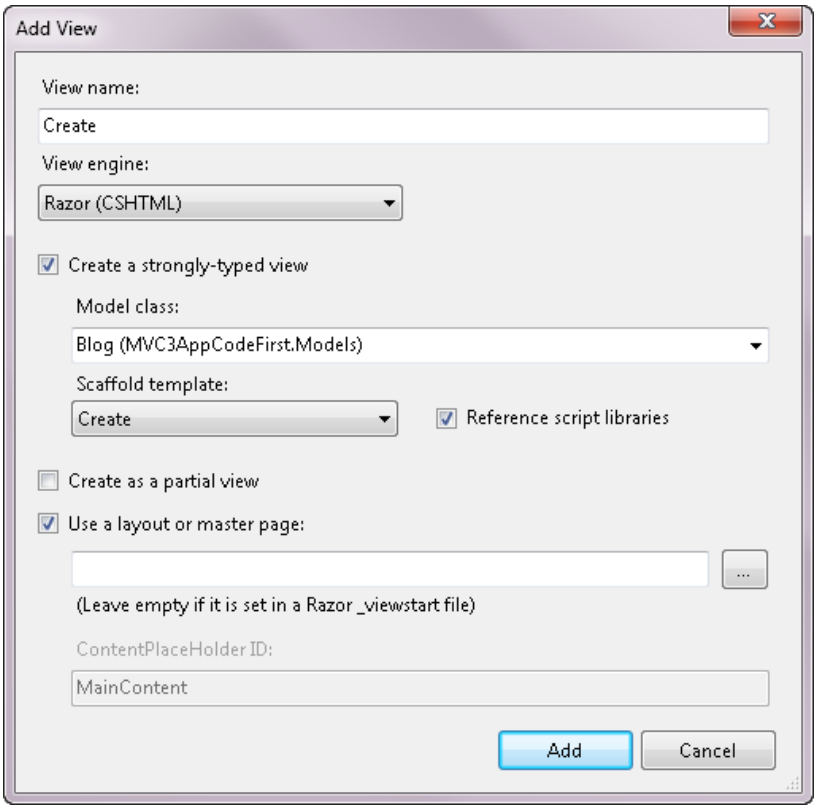

Figure 6: Creating a new view to allow users to add a new Blog.

The new view will be added to the Views/Blog folder along with the Index view you created.

Running the Application

When you first created the MVC application with the template defaults, Visual Studio created a global.asax file that has in it an instruction, called a routing, that tells the application to start with the Index view of the Home controller. You'll need to change that to start with the Index View of the Blog controller instead.

- Open the global.asax file from Solution Explorer.
- Modify the MapRoute call to change the value of controller from "Home" to "Blog".

```
routes.MapRoute(
  "Default", // Route name
  "{controller}/{action}/{id}", // URL with parameters
  new { controller = "Blog", action = "Index", id = UrlParameter.Optional }
```

Now when you run the application the BlogController.Index action will be the first method called. In turn it will execute the code, retrieving a list of Blogs and returning it into the default view (the view of the same name, Index). This is what you will see. Note that I've made minor changes to the page's header in the Shared/ _Layout.cshtml file and the heading of the Index View.

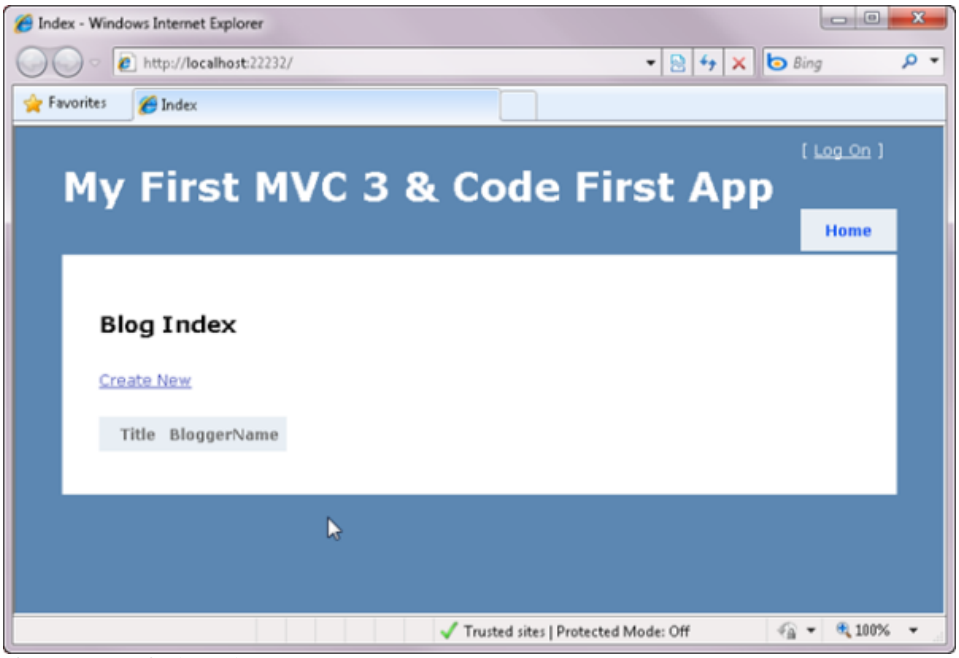

Figure 7: Blog List (empty)

Not much to see here except that when you return a View from the BlogController Index method, MVC will find the Blog Index view and display that along with whatever was passed in, which in this case was an empty list.

Clicking the Create New link will call the Create method which returns the Create view. After entering some information, such as that shown in Figure 8 and clicking the Create button, control is passed back to the Controller, this time calling the Create post back method.

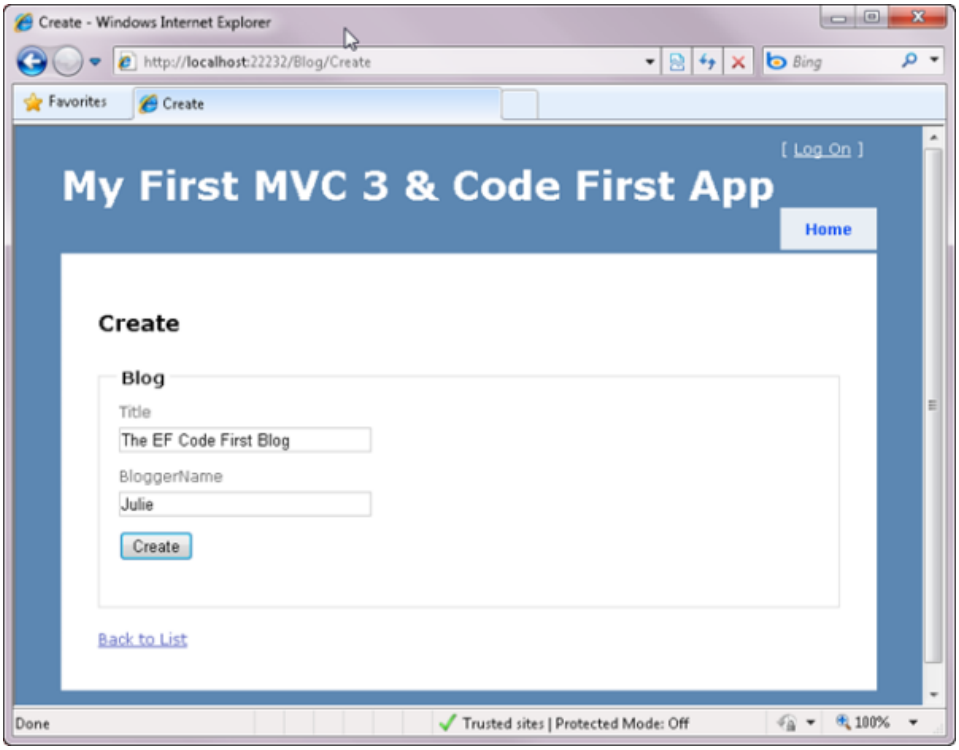

Figure 8: Creating a new blog

The view pushed a blog instance back to the Create method, which then uses the BlogContext to push that data into the database. The Create method then redirects to the Index method which re-queries the database and returns the Index View along with the list of Blogs. This time the list has a single item in it and is displayed as shown in Figure 9.

Figure 9: Blog list after adding new blog

Implementing Edit and Delete

Thus far you have queried for data and added new data. Now you'll see how Code First lets you edit and delete data just as easily.

To demonstrate this you'll use the BlogController's Edit and Delete methods. This means you'll need the corresponding Views.

1. Create an Edit View from the BlogController Edit method. Be sure to select a strongly typed view for the Blog class and the Edit scaffolding type.
2. Create a Delete View from the Blog Controller Edit method. Again, you must select a strongly typed view for the Blog class and this time, the Delete scaffolding type.

When a user edits a blog you will first need to retrieve that particular blog and then send it back to the Edit view. You can take advangate of Entity Framework 4.1's Find method which by default searches the entity's key property using the the value passed in. In this case, we'll find the Blog with an Id field that matches the value of id.

3. Modify the Edit method to match the following code;

```
public ActionResult Edit(int id)
{
     using (var db = new BlogContext())
     {
            return View(db.Blogs.Find(id));
     }
```

```
}
```

The job of the overloaded Edit method used for postbacks is to update the blog in the database with the data that came from the view. A single line of code that leverages the DbContext.Entry method will make the context aware of the Blog and then tell the context that this Blog has been modified. By setting the State to Modified, you are instructing Entity Framework that it should construct an UPDATE command when you call SaveChanges in the next line of code.

Your project will need a reference to the System.Data.Entity namespace in order to change the State. The first few steps that follow show you how to do that.

1. Right click the References folder in Solution Explorer.
2. Choose Add Reference… from the References context menu.
3. Select the .NET tab in the Add Reference dialog.
4. Locate the System.Data.Entity component, select it and then click OK.
5. Modify the Edit HttpPost method to make the context aware of the Blog and then save it back to the database as follows:

```
[HttpPost]
public ActionResult Edit(int id, Blog blog)
{
          try
          {
            context.Entry(blog).State = System.Data.EntityState.Modified;
            context.SaveChanges();
            return RedirectToAction("Index");
          }
          catch
          {
            return View();
          }
}
```

Notice that you just as you did for the Create method, you need to change the method signature to accept a Blog type instead of a FormCollection.

By default, the Edit view presents all of the properties except the key, BlogID.

Figure 10: Editing the blog information

After the changes are saved in the Edit postback method, MVC redirects back to the Index method. A fresh query is executed to retrieve the Blogs, including the newly updated Blog, which are then displayed in the Index view.

Figure 11: After editing blog information

Deleting a Blog will work in much the same way.

Modify the Delete methods to match the following examples:

```
public ActionResult Delete(int id)
{
      using (var context = new BlogContext())
      {
        return View(context.Blogs.Find(b));
      }
    }
    [HttpPost]
    public ActionResult Delete(int id, Blog blog)
    {
      try
      {
        using (var context = new BlogContext())
        {
          context.Entry(blog).State = System.Data.EntityState.Deleted;
          context.SaveChanges();
        }
        return RedirectToAction("Index");
      }
      catch
      {
        return View();
      }
}
```

The first Delete method is just the same as its Edit counterpart. And the postback is similar to the Edit postback with one exception: you are setting the State to Delete rather than to Modified.

The Delete view presents a confirmation screen to the user.

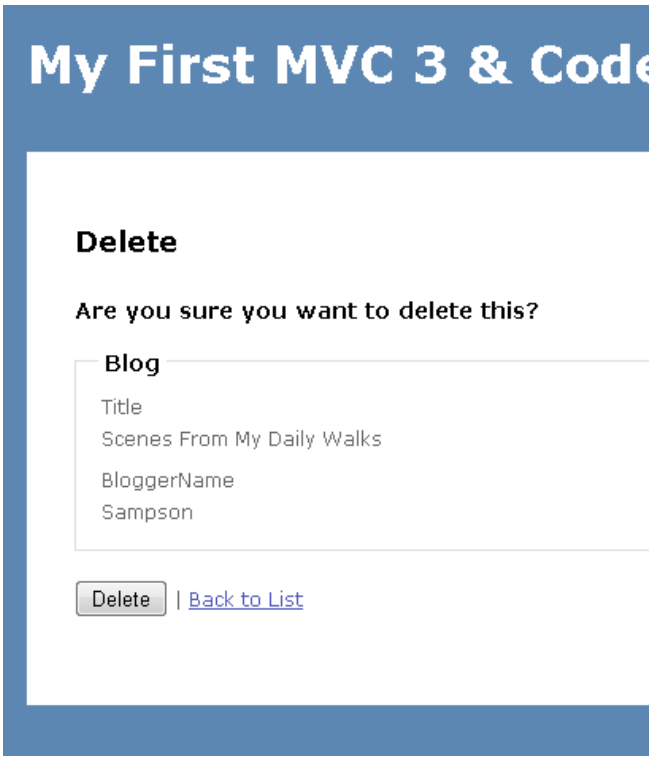

Figure 12: Delete Confirmation Screen

When the user clicks the Delete button, the Delete postback method will be executed and the user will be returned to the Index view.

Summary

Entity Framework's Code First enables the simplest path from your logic to your persisted data. At its simplest, you need no more than a reference to the special API and a class that derives from DbContext that has knowledge of your domain classes. Code First will even create the data store for you, by default, a SQL Express database with no effort on your part.

About the Author

Julie Lerman is a Microsoft MVP, .NET mentor and consultant who lives in the hills of Vermont. You can find her presenting on data access and other Microsoft .NET topics at user groups and conferences around the world. Julie blogs at thedatafarm.com/blog and is the author of the highly acclaimed book, "Programming Entity Framework" (O'Reilly Media, 2009). Follow her on Twitter.com: julielerman.