SOFTWARE GUILD

Entity Framework Code First

.NET Cohort

Coding Bootcamp

SOFTWARE GUILD

Lesson Goals

- Create a simple code-first model for a database that does not exist.
- Learn about code-first migrations

Create a Console Application

Let's create a console application and two classes: Blog and Post. A blog has a name and contains posts. A post has content and a foreign key to a blog.

Next, let's use NuGet to add Entity Framework to the project.

Then, we can create a DBContext class that contains a DbSet for each object we want to create and store in our database.

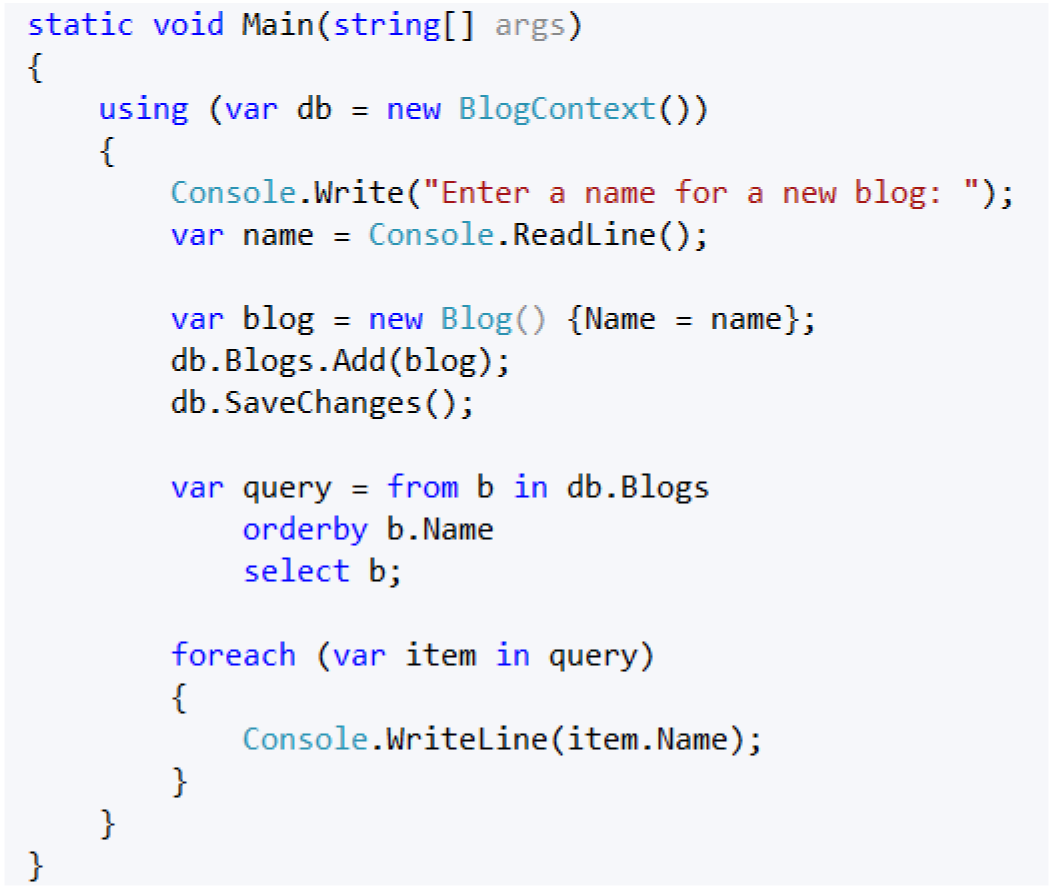
```
public class Blog
{
    public int BlogId { get; set; }
    public string Name { get; set; }

    public virtual List<Post> Posts { get; set; }
}

public class Post
{
    public int PostId { get; set; }
    public string Title { get; set; }
    public string Content { get; set; }

    public int BlogId { get; set; }
    public virtual Blog Blog { get; set; }
}


public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
}
```

Let’s Write Some Code

Now let’s write some code to add a new blog. You will notice that this code will run and work even though we haven’t set up any information about a database.

In Visual Studio 2012, it will create databases in the LocalDb, which can be accessed via the SQL Server Object Explorer window.

We may want to make some changes after testing, which will require us to enable migrations. Open the NuGet console and type in Enable-Migrations



```
static void Main(string[] args)
{
    using (var db = new BlogContext())
    {
        Console.Write("Enter a name for a new blog: ");
        var name = Console.ReadLine();

        var blog = new Blog() {Name = name};
        db.Blogs.Add(blog);
        db.SaveChanges();

        var query = from b in db.Blogs
            orderby b.Name
            select b;

        foreach (var item in query)
        {
            Console.WriteLine(item.Name);
        }
    }
}
```

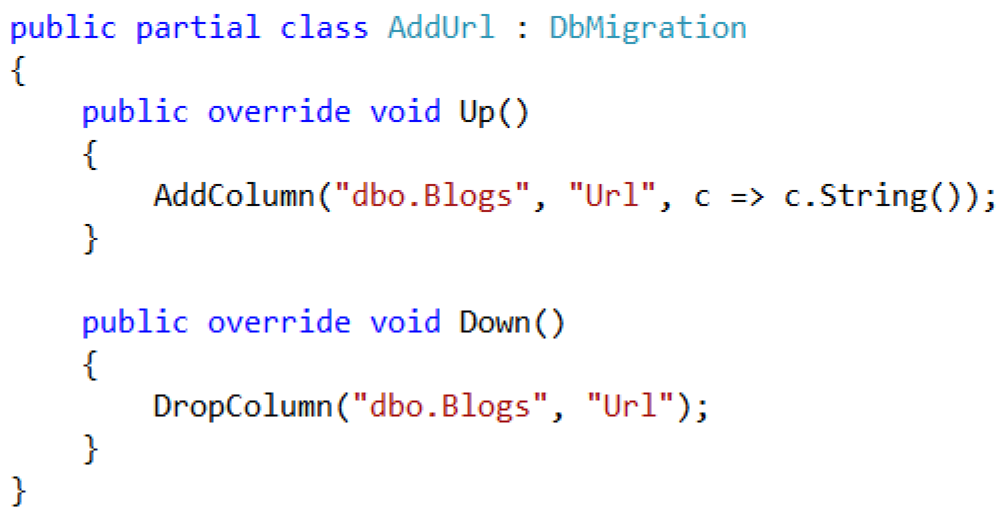
Migrations

Enable migrations creates a couple files for us. The first is a configuration file which allows us to do things like specify seed data that should be created in the database.

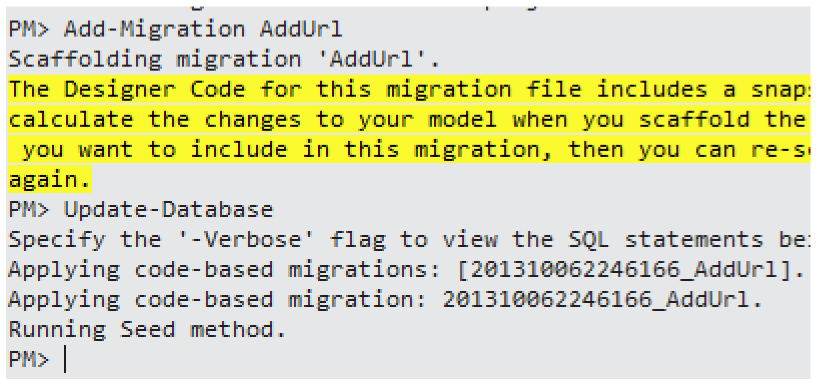
The second file is the code to set up our database tables. It has Up and Down methods for enabling a migration and rolling it back.

Let’s add a new property to blog called Url. In the console, we can Add-Migration AddUrl to create an Up and Down file for adding a new column.

Lastly, we can Update-Database in the console to run the migration.



```
public partial class AddUrl : DbMigration
{
    public override void Up()
    {
        AddColumn("dbo.Blogs", "Url", c => c.String());
    }

    public override void Down()
    {
        DropColumn("dbo.Blogs", "Url");
    }
}
```



```
PM> Add-Migration AddUrl
Scaffolding migration 'AddUrl'.
The Designer Code for this migration file includes a snap
calculate the changes to your model when you scaffold the
 you want to include in this migration, then you can re-s
again.
PM> Update-Database
Specify the '-Verbose' flag to view the SQL statements be
Applying code-based migrations: [201310062246166_AddUrl].
Applying code-based migration: 201310062246166_AddUrl.
Running Seed method.
PM> |
```

Data Annotations

We can customize the database structures created by adding data annotations to our classes.

Let’s create a class called User, where Username is the primary key, instead of an auto number.

To do this, we can put the [Key] annotation above the Username property. Add the User DbSet and then add a migration + update.

There are many Data Annotations, look them up on MSDN!

```
public class User
{
    [Key]
    public string Username { get; set; }
    public string DisplayName { get; set; }
}

public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }
}
```

Altering Schema With Fluent API

Another way to alter schema is to use the fluent API. For this, we override the OnModelCreating method in the DbContext.

From there, we can tell the modelBuilder that we would like the User entity's DisplayName property to have a max length of 25.

Add a migration and update the database and you will see the schema change is complete.

```
public class BlogContext : DbContext
{
    public DbSet<Blog> Blogs { get; set; }
    public DbSet<Post> Posts { get; set; }
    public DbSet<User> Users { get; set; }

    protected override void OnModelCreating(DbModelBuilder modelBuilder)
    {
        modelBuilder.Entity<User>()
            .Property(u => u.DisplayName)
            .HasMaxLength(25);
    }
}
```

Conclusion

- With code-first development, we can build object models that Entity Framework can turn into database tables.
- We need to be wary that Entity Framework will not automatically size or type properties in an efficient manner unless we use Data Annotations.