Entity Framework has become our ORM of choice lately and with their Code First feature our database access is now one of the simplest part of our projects. Creating your entire database schema by simply typing out a few C# classes allows you to get your project up and running quickly. But as developers we are always thinking of ways to make this process even easier. One improvement we have made to our infrastructure is to create a base class from which all of our entities inherit from. Here at Atmist, all entities we create contain the same three columns:

1. Id – An auto-incremented primary key (also known as a surrogate key)
2. CreatedDate – The DateTime when this entity was first created
3. ModifiedDate – The DateTime when this entity was last modified

Instead of adding these properties to every entity we create, let's use object inheritance to our advantage. In this article I will show you how to create a custom base class for Entity Framework Code First that you can use for all of the entities in your project and several benefits of taking this approach. Please note that if you are completely new to Entity Framework Code First I'd recommend following some beginner tutorials here.

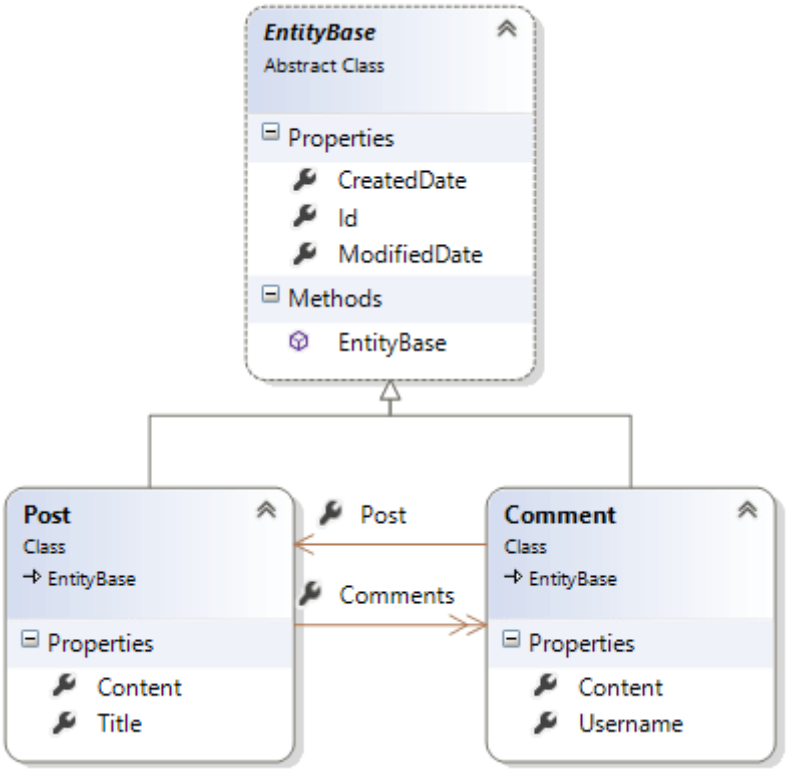
Entity Framework Inheritance

Entity Framework has three different ways to handle inheritance when mapping to SQL: Table per Hierarchy, Table per Type, and Table per Concrete Type. Feel free to read more about these three approaches but for this article we will be using the third type: Table per Concrete Type. Luckily for us this is also Entity Framework's default so we won't have any extra customization to do.

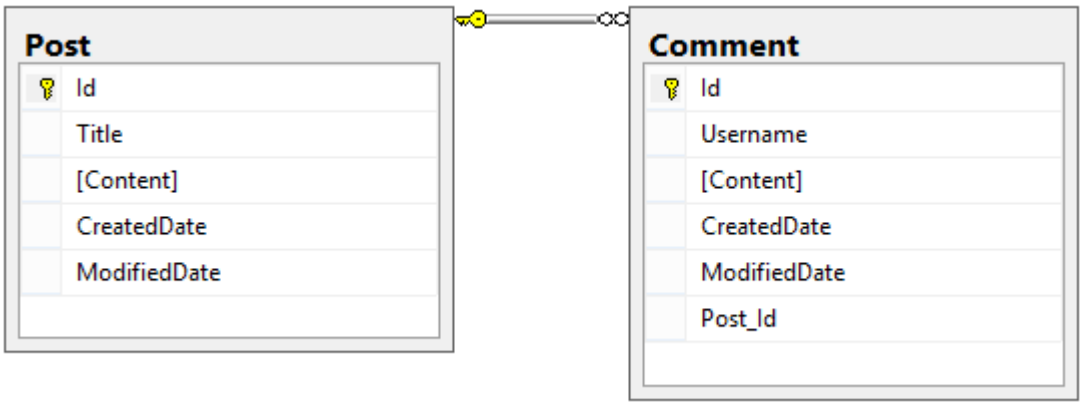
So let's jump right into it. The code below is what we use for our base class which includes the three properties I mentioned in the introduction.

```csharp
public abstract class EntityBase
{
    public EntityBase()
    {
        CreatedDate = DateTime.Now;
        ModifiedDate = DateTime.Now;
    }

    [Key]
    public int Id { get; set; }

    [Required]
    public DateTime CreatedDate { get; set; }

    [Required]
    public DateTime ModifiedDate { get; set; }
}
```

Pretty simple, the three fields I mentioned above are all there. Note the Key attribute on the Id property. This will tell Entity Framework to use an Identity type for this column. All entities we create will be subclasses of this entity base class. For example, if we were creating a revolutionary new blogging framework we may have entities for posts and comments. A simple UML diagram of the code first entities might look something like this:



Once we are happy with our entities we will then generate a migration using the Add-Migration command. This will create us a set of SQL tables similar to the following:



The big thing to note in the transformation from C# class to SQL table is that EntityBase doesn't get mapped to its own table. Instead EntityBase's properties get included in *every single one* of its subclasses.

Benefits of a base entity

Alright, I can hear you saying it now: "Cool diagrams but *why* would I use this?". Utilizing object inheritance in your code first entities has many benefits but the two big ones we will touch on in rest of this article are: generic extension methods and auto-updated date properties.

Generic extension methods on EntityBase

Now that we have a base class for all of our entities we can leverage C# generics and create some handy extension methods that will apply to all collections of entities. Here are a few that we have found useful:

```C#
public static class EntityBaseQueries
{
    public static TEntity GetById<TEntity>(this IQueryable<TEntity> set,
int id) where TEntity : EntityBase
    {
        return set.SingleOrDefault(x => x.Id == id);
    }

    public static IQueryable<TEntity> GetByIds<TEntity>(this
IQueryable<TEntity> set, List ids) where TEntity : EntityBase
    {
        return set.Where(x => ids.Contains(x.Id));
    }

    public static TEntity Remove<TEntity>(this IDbSet<TEntity> set, int id)
where TEntity : EntityBase
    {
        var entity = set.GetById(id);
        DBEntities.Current.Remove(entity);
        return entity;
    }

    public static IQueryable<TEntity> OrderByRecentlyCreated<TEntity>(this
IDbSet<TEntity> set) where TEntity : EntityBase
    {
        return set.OrderByDescending(x => x.CreatedDate);
    }
}
```

Since these extension methods are generic on EntityBase they are available on all of your collections of entities. This makes it very easy to quickly load up entities by Id or sort any collections based on created or modified dates. Speaking of dates…

Automatically updated properties

One of the biggest benefits we have found is automatically updating our CreatedDate and ModifiedDate fields. Doing this manually whenever you create or edit an entity is tedious and is usually forgotten altogether. Thankfully, Entity Framework provides an easy way to intercept saving and easily check which entities have changed. By overriding the SaveChanges method in your DbContext subclass we can intercept when entities are saved and update their properties. Below is an example of how we automatically update our dates:

```C#
public override int SaveChanges()
{
    foreach (var entry in ChangeTracker.Entries())
    {
        var entity = entry.Entity;
        if (entry.State == EntityState.Added)
        {
            entity.CreatedDate = DateTime.Now;
            entity.ModifiedDate = DateTime.Now;
        }
        else if (entry.State == EntityState.Modified)
        {
            entity.ModifiedDate = DateTime.Now;
        }
    }
    return base.SaveChanges();
}
```

By utilizing the ChangeTracker on the database context we can retrieve all the entities that have changed. If an entity has been added we set its created date and modified date. If the entity is just being updated, we only change its modified date. Our date columns on *all* entities are now automatically updated every time we create or edit entities.

Conclusion

Not only is Entity Framework Code First an easy way to create your entities at the start of your project, it also provides great features that allow you to streamline your database access. By utilizing a base class for all of our entities we gained much needed consistency across our database objects. Combined with generic extension methods and automatically updated columns most of the tediousness of database access just disappears. If you have any any questions, comments, or your own useful extension methods, let us know in the comments.