• 15 minutes to read

April 2018

Volume 33 Number 4

[Data Points]

EF Core 2 Owned Entities and Temporary Work-Arounds

By Julie Lerman

The new Owned Entity feature in EF Core 2.0 replaces the Complex Type feature of Entity Framework "classic" (EF thru EF6). Owned Entities allow the mapping of value objects to the data store. It's quite common to have a business rule that permits properties based on value objects to be null. Moreover, because value objects are immutable, it's also important to be able to replace properties that contain a value object. The current version of EF Core doesn't allow either of these scenarios by default, although both will be supported in upcoming iterations. In the meantime, rather than treating these limitations as showstoppers for those of us who love the benefits of domain-driven design (DDD) patterns, this article will show you how to work around those limitations. A DDD practitioner may still reject these temporary patterns as not following the principles of DDD closely enough, but the pragmatist in me is satisfied, buttressed by the knowledge that they are simply temporary solutions.

Notice that I said "by default." It turns out that there is a way to have EF Core take responsibility to enforce its own rule about null owned entities and allow for value object replacement, without dramatically affecting your domain classes or your business rules. In this column, I'll show you how to do this.

There is another way around the nullability problem, which is to simply map the value object type to its own table, thus physically splitting the value object data from the rest of the object to which it belongs. While this may be a good solution for some scenarios, it's generally one I don't want to use. Therefore, I prefer to use my work-around, which is the focus of this column. But first I want to ensure you understand why this problem and solution are important enough that I'm devoting this column to the topic.

A Short Primer on Value Objects

Value objects are a type that lets you encapsulate multiple values into a single property. A string is a great example of a value object. Strings are made up of a collection of chars. And they are immutable—immutability is a critical facet of a value object. The combination and order of the letters c, a and r have a specific meaning. If you were to mutate it, for example by changing the last letter to a "t," you'd completely change the meaning. The object is defined by the combination of all of its values. As such, the fact that the object can't be modified is part of its contract. And there's another important aspect of a value object—it doesn't have its own identity. It can be used only as a property of another class, just like a string. Value objects have other contractual rules, but these are the most important ones to start with if you're new to the concept.

Given that a value object is composed of its properties and then, as a whole, used as a property in another class, persisting its data takes some special effort. With a non-relational database such as a document database, it's easy to just store the graph of an object and its embedded value objects. But that's not the case when storing into a relational database. Starting with the very first version, Entity Framework included the ComplexType, which knew how to map the properties of the property to the database where EF was persisting your data. A common value object example is PersonName, which might consist of a FirstName property and a LastName property. If you have a Contact type with a PersonName property, by default, EF Core will store the FirstName and LastName values as additional columns in the table to which Contact is mapped.

An Example of a Value Object in Use

I've found that looking at a variety of examples of value objects helped me to better understand the concept, so, I'll use yet another example—a SalesOrder entity and a PostalAddress value object. An order typically includes both a shipping address and a billing address. While those addresses may exist for other purposes, within the context of the order, they're an integral part of its definition. If a person moves to a new location, you still want to know where that order was shipped, so it makes sense to embed the addresses into the order. But in order to treat addresses consistently in my system, I prefer to encapsulate the values that make up an address in their own class, PostalAddress, as shown in **Figure 1**.

Figure 1 PostalAddress ValueObject

```
public class PostalAddress : ValueObject<PostalAddress>
{
  public static PostalAddress Create (string street, string city,
                                      string region, string postalCode)   {
    return new PostalAddress (street, city, region, postalCode);
  }
  private PostalAddress () { }
  private PostalAddress (string street, string city, string region,
                         string postalCode)   {
    Street = street;
    City = city;
    Region = region;
    PostalCode = postalCode;
  }
  public string Street { get; private set; }
  public string City { get; private set; }
  public string Region { get; private set; }
  public string PostalCode { get; private set; }
  public PostalAddress CopyOf ()   {
    return new PostalAddress (Street, City, Region, PostalCode);
  }
}
```

PostalAddress inherits from a ValueObject base class created by Jimmy Bogard (bit.ly/2EpKydG). ValueObject provides some of the obligatory logic required of a value object. For example, it has an override of Object.Equals, which ensures that every property is compared. Keep in mind that it makes heavy use of reflection, which may impact performance in a production app.

Two other important features of my PostalAddress value object are that it has no identity key property and that its constructor forces the invariant rule that every property must be populated. However, for an owned entity to be able to map a type defined as a value object, the only rule is that it have no identity key of its own. An owned entity is not concerned with the other attributes of a value object.

With PostalAddress defined, I can now use it as the Shipping Address and BillingAddress properties of my SalesOrder class (see **Figure 2**). They aren't navigation properties to related data, but just more properties similar to the scalar Notes and OrderDate.

Figure 2 The SalesOrder Class Contains Properties That Are PostalAddress Types

```
public class SalesOrder {
  public SalesOrder (DateTime orderDate, decimal orderTotal)   {
    OrderDate = orderDate;
    OrderTotal = orderTotal;
    Id = Guid.NewGuid ();
  }
  private SalesOrder () { }
  public Guid Id { get; private set; }
  public DateTime OrderDate { get; private set; }
  public decimal OrderTotal { get; private set; }
  private PostalAddress _shippingAddress;
  public PostalAddress ShippingAddress => _shippingAddress;
  public void SetShippingAddress (PostalAddress shipping)
  {
    _shippingAddress = shipping;
  }
  private PostalAddress _billingAddress;
  public PostalAddress BillingAddress => _billingAddress;
  public void CopyShippingAddressToBillingAddress ()
  {
    _billingAddress = _shippingAddress?.CopyOf ();
  }
  public void SetBillingAddress (PostalAddress billing)
  {
    _billingAddress = billing;
  }
}
```

These addresses now live within the SalesOrder and can provide accurate information regardless of the current address of the person who placed the order. I will always know where that order went.

Mapping a Value Object as an EF Core Owned Entity

In earlier versions, EF could automatically recognize classes that should be mapped using a ComplexType by discovering that the class was used as a property of another entity and it had no key property. EF Core, however, can't automatically infer owned entities. You must specify this in the DbContext Fluent API mappings in the OnModelCreating method using the new OwnsOne method to specify which property of that entity is the owned entity:

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
  modelBuilder.Entity<SalesOrder>().OwnsOne(s=>s.BillingAddress);
  modelBuilder.Entity<SalesOrder>().OwnsOne(s=>s.ShippingAddress);
}
```

I've used EF Core migrations to create a migration file describing the database to which my model maps. **Figure 3** shows the section of the migration that represents the SalesOrder table. You can see that EF Core understood that the properties of PostalAddress for each of the two addresses are part of the SalesOrder. The column names are as per EF Core convention, although you can affect those with the Fluent API.

Figure 3 The Migration for the SalesOrder Table, Including All of the Columns for PostalAddress Properties

```
migrationBuilder.CreateTable(
  name: "SalesOrders",
  columns: table => new
  {
    Id = table.Column(nullable: false)
              .Annotation("Sqlite:Autoincrement", true),
    OrderDate = table.Column(nullable: false),
    OrderTotal = table.Column(nullable: false),
    BillingAddress_City = table.Column(nullable: true),
    BillingAddress_PostalCode = table.Column(nullable: true),
    BillingAddress_Region = table.Column(nullable: true),
    BillingAddress_Street = table.Column(nullable: true),
    ShippingAddress_City = table.Column(nullable: true),
    ShippingAddress_PostalCode = table.Column(nullable: true),
    ShippingAddress_Region = table.Column(nullable: true),
    ShippingAddress_Street = table.Column(nullable: true)
  }
```

Additionally, as mentioned earlier, putting the addresses into the SalesOrder table is by convention, and my preference. This alternate code will split them out to separate tables and avoid the nullability problem completely:

```
modelBuilder.Entity<SalesOrder> ().OwnsOne (
  s => s.BillingAddress).ToTable("BillingAddresses");
modelBuilder.Entity<SalesOrder> ().OwnsOne (
  s => s.ShippingAddress).ToTable("ShippingAddresses");
```

Creating a SalesOrder in Code

Inserting a sales order with both the billing address and shipping address is simple:

```
private static void InsertNewOrder()
{
  var order=new SalesOrder{OrderDate=DateTime.Today, OrderTotal=100.00M};
  order.SetShippingAddress (PostalAddress.Create (
    "One Main", "Burlington", "VT", "05000"));
  order.SetBillingAddress (PostalAddress.Create (
    "Two Main", "Burlington", "VT", "05000"));
  using(var context=new OrderContext()){
    context.SalesOrders.Add(order);
    context.SaveChanges();
  }
}
```

But let’s say that my business rules allow an order to be stored even if the shipping and billing address haven’t yet been entered, and a user can complete the order at another time. I’ll comment out the code that fills the BillingAddress property:

```
// order.BillingAddress=new Address("Two Main","Burlington", "VT", "05000");
```

When SaveChanges is called, EF Core tries to figure out what the properties of the BillingAddress are so that it can push them into the SalesOrder table. But it fails in this case because BillingAddress is null. Internally, EF Core has a rule that a conventionally mapped owned type property can’t be null.

EF Core is assuming that the owned type is available so that its properties can be read. Developers may see this as a showstopper for being able to use value objects or, worse, for being able to use EF Core, because of how critical value objects are to their software design. That was how I felt at first, but I was able to create a work-around.

Temporary Work-Around to Allow Null Value Objects

The goal of the work-around is to ensure that EF Core will receive a ShippingAddress, BillingAddress or other owned type, whether or not the user supplied one. That means the user isn't forced to supply a shipping or billing address just to satisfy the persistence layer. If the user doesn't supply one, then a PostalAddress object with null values in its properties will be added in by the DbContext when it's time to save a SalesOrder.

I made a minor adaptation to the PostalAddress class by adding a second factory method, Empty, to let the DbContext easily create an empty PostalAddress:

```
public static PostalAddress Empty()
{
  return new PostalAddress(null,null,null,null);
}
```

Additionally, I enhanced the ValueObject base class with a new method, IsEmpty, shown in **Figure 4**, to allow code to easily determine if an object has all null values in its properties. IsEmpty leverages code that already exists in the ValueObject class. It iterates through the properties and if any one of them has a value, it returns false, indicating that the object is not empty; otherwise, it returns true.

Figure 4 The IsEmpty Method Added to the ValueObject Base Class

```
public bool IsEmpty ()
{
  Type t = GetType ();
  FieldInfo[] fields = t.GetFields
    (BindingFlags.Instance | BindingFlags.NonPublic | BindingFlags.Public);
  foreach (FieldInfo field in fields)
  {
    object value = field.GetValue (this);
    if (value != null)
    {
      return false;
    }
  }
  return true;
}
```

But my solution for allowing null owned entities wasn't yet complete. I still needed to use all of this new logic to ensure that new SalesOrders would always have a ShippingAddress and a BillingAddress in order for EF Core to be able to store them into the database. When I initially added this last piece of my solution, I wasn't happy with it because that last bit of code (which I won't bother sharing) was making the SalesOrder class enforce EF Core's rule—the bane of domain-driven design.

Voila! An Elegant Solution

Luckily, I was speaking at DevIntersection, as I do every fall, where Diego Vega and Andrew Peters from the EF team were also presenting. I showed them my work-around and explained what was bothering me—the need to enforce non-null ShippingAddress and BillingAddress in the SalesOrder—and they agreed. Andrew quickly came up with a way to use the work I had done in the ValueObject base class and the tweak I made to the PostalAddress to force EF Core to take care of the problem without putting the onus on SalesOrder. The magic happens in the override of the SaveChanges method of my DbContext class, shown in **Figure 5**.

Figure 5 Overriding SaveChanges to Provide Values to Null Owned Types

```
public override int SaveChanges()
{
  foreach (var entry in ChangeTracker.Entries()
             .Where(e => e.Entity is SalesOrder && e.State == EntityState.Addec
  {
    if (entry.Entity is SalesOrder)
    {
      if (entry.Reference("ShippingAddress").CurrentValue == null)
      {
        entry.Reference("ShippingAddress").CurrentValue = PostalAddress.Empty(
      }
      if (entry.Reference("BillingAddress").CurrentValue == null)
      {
        entry.Reference("BillingAddress").CurrentValue = PostalAddress.Empty()
      }
  }
  return base.SaveChanges();
}
```

From the collection of entries that the DbContext is tracking, SaveChanges will iterate through those that are SalesOrders flagged to be added to the database, and will make sure they get populated as their empty counterparts.

What About Querying Those Empty Owned Types?

Having satisfied the need for EF Core to store null value objects, it's now time to query them back from the database. But EF Core resolves those properties in their empty state. Any ShippingAddress or BillingAddress that was originally null comes back as an instance with null values in its properties. After any query, I need my logic to replace any empty PostalAddress properties with null.

I spent quite some time looking for an elegant way to achieve this. Unfortunately, there isn't yet a lifecycle hook to modify objects as they're being materialized from query results. There's a replaceable service in the query pipeline called CreateReadValueExpression in the internal EntityMaterializerSource class, but that can only be used on scalar values, not objects. I tried numerous other approaches that were more and more complicated, and finally had a long talk with myself about the fact that this is a temporary workaround, so I can accept a simpler solution even if it is has a little bit of code smell. And this task isn't too difficult to control if your queries are encapsulated in a class dedicated to making EF Core calls to the database.

I named the method FixOptionalValueObjects:

```
private static void FixOptionalValueObjects (SalesOrder order) {
  if (order.ShippingAddress.IsEmpty ()) { order.SetShippingAddress (null); }
  if (order.BillingAddress.IsEmpty ()) { order.SetBillingAddress (null); }
}
```

Now I have a solution in which the user can leave value objects null and let EF Core store and retrieve them as non-nulls, yet return them to my code base as nulls anyway.

Replacing Value Objects

I mentioned another limitation in the current version of EF Core 2, which is the inability to replace owned entities. Value objects are by definition immutable. So, if you need to change one, the only way is to replace it. Logically this means that you're modifying the SalesOrder, just as if you had changed its OrderDate property. But because of the way that EF Core tracks the owned entities, it will always think the replacement is added, even if its host, SalesOrder, for example, is not new.

I made a change to the SaveChanges override to fix this problem (see **Figure 6**). The override now filters for SalesOrders that are added or modified, and with the two new lines of code that modify the state of the reference properties, ensures that ShippingAddress and BillingAddress have the same state as the order—which will either be Added or Modified. Now modified SalesOrder objects will also now be able to include the values of the ShippingAddress and BillingAddress properties in their UPDATE commands.

Figure 6 Making SaveChanges Comprehend Replaced Owned Types by Marking Them as Modified

```
public override int SaveChanges () {
  foreach (var entry in ChangeTracker.Entries ().Where (
    e => e.Entity is SalesOrder &&
    (e.State == EntityState.Added || e.State == EntityState.Modified))) {
    if (entry.Entity is SalesOrder order) {
      if (entry.Reference ("ShippingAddress").CurrentValue == null) {
        entry.Reference ("ShippingAddress").CurrentValue = PostalAddress.Empty
      }
      if (entry.Reference ("BillingAddress").CurrentValue == null) {
        entry.Reference ("BillingAddress").CurrentValue = PostalAddress.Empty (
      }
      entry.Reference ("ShippingAddress").TargetEntry.State = entry.State;
      entry.Reference ("BillingAddress").TargetEntry.State = entry.State;
    }
  }
  return base.SaveChanges ();
}
```

This pattern works because I'm saving with a different instance of OrderContext than I queried, which therefore doesn't have any preconceived notion of the state of the PostalAddress objects. You can find an alternate pattern for tracked objects in the comments of the GitHub issue at bit.ly/2sxMECT.

Pragmatic Solution for the Short-Term

If changes to allow optional owned entities and replacing owned entities were not on the horizon, I'd most likely take steps to create a separate data model to handle the data persistence in my software. But this temporary solution saves me that extra effort and investment and I know that soon I can remove my work-arounds and easily map my domain models directly to my database letting EF Core define the data model. I was happy to invest the time, effort and thought into coming up with the work-arounds so I can use value objects and EF Core 2 when designing my solutions—and help others be able to do the same.

Note that the download that accompanies this article is housed in a console app to test out the solution as well as persist the data to a SQLite database. I'm using the database rather than just writing tests with the InMemory provider because I wanted to inspect the database to be 100 percent sure that the data was getting stored the way I expected.

Julie Lerman *is a Microsoft Regional Director, Microsoft MVP, software team coach and consultant who lives in the hills of Vermont. You can find her presenting on data access and other topics at user groups and conferences around the world. She blogs at the thedatafarm.com/blog and is the author of "Programming Entity Framework," as well as a Code First and a DbContext edition, all from O'Reilly Media. Follow her on Twitter: @julielerman and see her Pluralsight courses at bit.ly/PS-Julie.*

Thanks to the following Microsoft technical expert for reviewing this article: Andriy Svyryd
Andriy Svyryd is a Ukrainian .NET developer who has worked on the Entity Framework team since 2010. See all the projects he contributes to at github.com/AndriySvyryd.

Discuss this article in the MSDN Magazine forum