Modeling Aggregates with DDD and Entity Framework

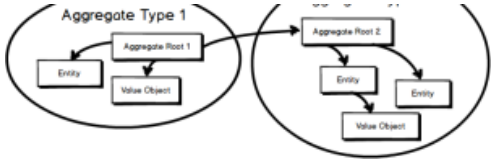
By Vaughn Vernon October 13, 2014(https://kalele.io/2014/10/13/)

For everyone who has read my book (https://kalele.io/books/) and/or *Effective Aggregate Design,* but have been left wondering how to implement Aggregates with Domain-Driven Design (DDD) on the .NET platform using C# and Entity Framework, this post is for you.

[NOTE: As expected, this article has within hours of posting received some criticism for the approach used to O-R mapping with Entity Framework. Actually the article received much more praise than criticism, but... I want to just point out that I am purposely not attempting to win any guru award in Entity Framework mapping. If you browse through this post too quickly some of the key words of wisdom and my intent may be lost on your speed reading. I am purposely avoiding some of the expert guidance that is typically given with a view to deep understanding of Entity Framework mappings. In fact, you may not realize the purpose of the article unless you begin reading with the assumed attitude that "I hate O-R mapping." The O-R mapping tooling is actually something like 20+ years old, and it is time that we come up with more practical solutions to storing objects as objects. In the meantime we should just do as little O-R mapping as we can get away with. So, thanks for your words of advice, but I have done everything below with precise intent.]

Definition of Aggregate

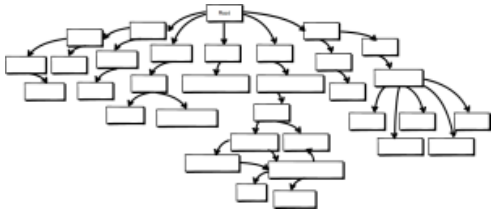
To start off, let's recap the basic definition of DDD Aggregate. First and foremost the Aggregate pattern is about transactional consistency. At the end of a committed database transaction, a single Aggregate should be completely up to date. That means that any business rules regarding data consistency must be met and the persistence store should hold that consistent state, leaving the Aggregate correct and ready to use by the next use case. Figure 1 illustrates two such consistency boundaries, with two different Aggregates.



(https://kalele.io/wp-content/uploads/2014/10/Aggregates.png)

Figure 1. Two Aggregates, which represent two transactional consistency boundaries.

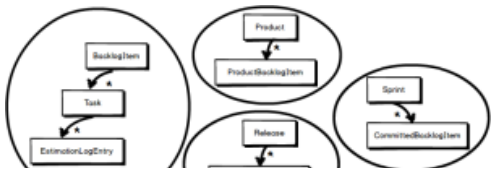
The problem that many have with designing Aggregates is that they don't consider the true business constraints that require data to be transactionally consistent and instead design Aggregates in large clusters as shown in Figure 2. Designing Aggregates in this way is a big mistake if you expect them (1) to be used by many thousands of users, (2) to perform well, and (3) to scale to the demands of the Internet.



(https://kalele.io/wp-content/uploads/2014/10/LargeCluster.png)

Figure 2. A poorly designed Aggregate that is not conceived on according to true business consistency constraints.

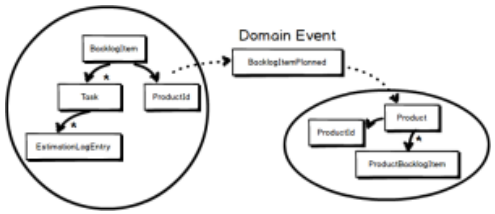
Using an example from my book (https://kalele.io/books/), a set of well-designed Aggregates are shown in Figure 3. These are based on true business rules that require specific data to be up-to-date at the end of a successful database transaction. These follow the rules of Aggregate, including *designing small Aggregates.*



(https://kalele.io/wp-content/uploads/2014/10/FourSmallAggregates.png)

Figure 3. Some well-designed Aggregates that adhere to true consistency rules.

Still, the question arises, if BacklogItem and Product have some data dependencies, how do we update both of them. This points to the another rule of Aggregate design, to *use eventual consistency* as shown in Figure 4. Of course, there's a bit more involved when you consider the overall architecture, but the foregoing points out the high-level composition guidance of Aggregate design.



(https://kalele.io/wp-content/uploads/2014/10/EventualConsistency.png)

Figure 4. When two or more Aggregates have at least some dependencies on updates, use eventual consistency.

Now with this brief refresher on the basics of Aggregate design, let's see how we might map the Product to a database using Entity Framework.

KISS with Entity Framework

So, we have four prominent Aggregates in our Scrum project management application: Product, BacklogItem, Release, and Sprint. We need to persist the state of these four small Aggregates and we want to use Entity Framework to do so. Here's a possible surprise for you. I am not going to recommend that you need to become an Entity Framework guru. Nope, just the opposite in fact.

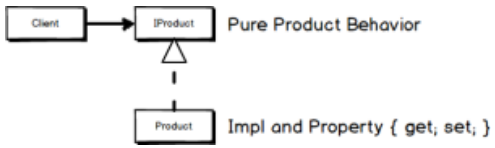
I am going to suggest that you allow the Entity Framework development team to be the gurus, and you just focus on your specific application. After all, your Core Domain is where you want to put your creative energies, not in becoming an expert in Entity Framework.

What I am recommending is that you allow Entity Framework to take control of doing what it does best and we just stay out of its way. Entity Framework has a certain way of mapping entities into the database, and that's just how it works. As soon as you try to step outside the basics and go to some extremes of esoteric mapping techniques in ways that Entity Framework was not meant to be used, you are going to experience a lot of pain. Still, we can get quite a bit of mileage out of Entity Framework in the midst of DDD and be quite happy with the way it all works out. To do so we are going to use just a few basic mapping techniques. If you follow my KISS guidance you can mostly ignore your Entity Framework documentation and how-to books. Just allow Entity Framework to map entities and get back to what will make a difference in this competitive world: *your market-distinguishing application.*

We are going to implement the Product Aggregate using two approaches. One approach uses a Separated Interface with an implementation class, and the other uses a domain object backed by a state object. The whole point of these examples is to stay as far out of Entity Framework's way as possible.

Using a Separated Interface and Implementation Class

For the first example I create a Separated Interface that is implemented by a concrete domain object. Figure 5 shows you the basic intention of this approach.



(https://kalele.io/wp-content/uploads/2014/10/EntityFramework1.png)

Figure 5. The Separated Interface named IProduct is implemented by a concrete domain object. Clients directly use only IProduct.

It is pretty typical when programming with C# and .NET to name your interfaces with an "I" prefix, so we will use IProduct:

```
interface IProduct
{
  ICollection<IBacklogItem> AllBacklogItems();
  IProductBacklogItem BacklogItem(BacklogItemId backlogItemId);
```

```
  string Description { get; }
  string Name { get; }
  IBacklogItem PlanBacklogItem(BacklogItemId newBacklogItemId, string summary,
      string story, string category, BacklogItemType type, StoryPoints storyPoints);
  void PlannedProductBacklogItem(IBacklogItem backlogItem);
  ...
  ProductId ProductId { get; }
  ProductOwnerId ProductOwnerId { get; }
  void ReorderFrom(BacklogItemId id, int ordering);
  TenantId TenantId { get; }
}
```

With this interface we can create a concrete implementation class. Let's call it Product:

```
public class Product : IProduct
{
  [Key]
  public string ProductKey { get; set; }
  ...
}
```

The point of the concrete class Product is to implement the business interface declared by IProduct and to also provide the accessors that are needed by Entity Framework to map the object into and out of the database. Note the ProductKey property. This is technically the kind of primary key that Entity Framework wants to work with. However, it is different from the ProductId, which when combined with the TenantId is the business identity. Therefore, internally the ProductKey must be set to a composite of TenantId as a string and ProductId as a string:

```
ProductKey = TenantId.Id + ":" + ProductId.Id;
```

I think you get the idea. We create an interface that we want our client to see and we hide the implementation details inside the

I think you get the idea. We create an interface that we want our client to see and we hide the implementation details inside the implementing class. We make the implementation match up to really basic Entity Framework mappings. We purposely try to keep our special mappings, as with ProductKey, to a minimum. This helps keep the DbContext very simple by registering the implementation classes:

```
public class AgilePMContext : DbContext
{
  public DbSet<Product> Products { get; set; }
  public DbSet<ProductBacklogItem> ProductBacklogItems { get; set; }
  public DbSet<BacklogItem> BacklogItems { get; set; }
  public DbSet<Task> Tasks { get; set; }
  ...
}
```

Rather than fully fleshing out the details of this approach, there is enough detail already to make some judgments. I'd like to discuss the fundamental flaws that I see in it:

- The Ubiquitous Language is not really reinforced by using interfaces such as IProduct, IBacklogItem, etc. IProduct and IBacklogItem are not in our Ubiquitous Language, but Product and BacklogItem are. Thus, the client facing names should be Product, BacklogItem, and the like. We could accomplish this simply by naming the interfaces Product, BacklogItem, Release, and Sprint, but that would mean we would have to come up with sensible names for the implementation classes. Let's just pause there and move on to the second and related issue.
- There is really no good reason to create a Separated Interface. It would be very unlikely that we would ever create two or more implementations of IProduct or any of the other interfaces. The best reason we have for creating a Separated Interface is when there could be or are multiple implementations, and is just not going to happen in this Core Domain.

Based on these two points alone I would personally choose to abandon this approach before going any further with it. When using Domain-Driven Design the most important and overarching principle is the adhere to the Ubiquitous Language, and from the get-go this approach is driving us away from business terminology rather than toward it.

Domain Object Backed By a State Object

The second approach uses a domain object backed by state objects. As shown in Figure 6, the domain object defines and implements the domain-driven model using the Ubiquitous Language, and the state objects hold the state of the Aggregate.

(https://kalele.io/wp-content/uploads/2014/10/EntityFramework2.png)

Figure 6. The domain object that models the Aggregate behavior is backed by a state object that holds the model's state.

By keeping state objects separate from the domain-driven implementation objects, it enables very simple mappings. We let Entity Framework to do what it knows how to do by default to map entities to and from the database. Consider Product, which is backed by the ProductState object. We have two Product constructors; a public business constructor for normal clients and a second internal constructor that is used only by internal implementation components:

```
public class Product
{
  public Product(
      TenantId tenantId,
```

```
      ProductId productId,
      ProductOwnerId productOwnerId,
      string name,
      string description)
  {
    State = new ProductState();
    State.ProductKey = tenantId.Id + ":" + productId.Id;
    State.ProductOwnerId = productOwnerId;
    State.Name = name;
    State.Description = description;
    State.BacklogItems = new List<ProductBacklogItem>();
  }

  internal Product(ProductState state)
  {
    State = state;
  }
  ...
}
```

When the business constructor is invoked we create a new ProductState object and initialize it. The state object has a simple string-based identity:

```
public class ProductState
{
  [Key]
  public string ProductKey { get; set; }
```

```
  public string ProductKey { get; set; }

  public ProductOwnerId ProductOwnerId { get; set; }

  public string Name { get; set; }

  public string Description { get; set; }

  public List<ProductBacklogItemState> BacklogItems { get; set; }
  ...
}
```

The ProductKey is actually encoded with two properties, the TenantId as a string and the ProductId as a string, with the two separated by a ':' character. Including the TenantId in the ProductKey ensures that all data stored in the database is segregated by tenant. We must still support client requests for TenantId and ProductId from the Product:

```
public class Product
{
  ...
  public ProductId ProductId { get { return new ProductId(State.DecodeProductId()); } }
  ...
  public TenantId TenantId { get { return new TenantId(State.DecodeTenantId()); } }
  ...
}
```

The ProductState object must support both DecodeProductId() and DecodeTenantId() methods. We could also choose to design the state object to redundantly hold whole identities separate of the ProductKey:

```
public class ProductState
{
  [Key]
  public string ProductKey { get; set; }
```

```
  public ProductId ProductId { get; set; }

  public ProductOwnerId ProductOwnerId { get; set; }

  public string Name { get; set; }

  public string Description { get; set; }

  public List<ProductBacklogItemState> BacklogItems { get; set; }

  public TenantId TenantId { get; set; }
  ...
}
```

This could be well worth the slight memory overhead if converting to identities had a heavy performance footprint. All of the identity types, including ProductOwnerId, are Value Objects and are flattened and mapped into the same database row that ProductState occupies:

```
[ComplexType]
public class ProductOwnerId : Identity
{
  public ProductOwnerId()
```

```
    public ProductOwnerId()
        : base()
    {
    }

    public ProductOwnerId(string id)
        : base(id)
    {
    }
}
```

The [ComplexType] attribute marks the Value Object as a complex type, which is different from an entity. Complex types are non-scalar values that do not have keys and cannot be managed apart from their containing entity, or the complex type within which they are nested. Marking a Value Object with the Entity Framework [ComplexType] causes the data of the Value Object to be saved to the same database row as the entity. In this case, ProductOwnerId would be saved to the same database row as the ProductState entity.

Here are the base types for all Identity types of Value Objects:

```
public abstract class Identity : IEquatable<Identity>, IIdentity
{
    public Identity()
    {
```

```
    {
      this.Id = Guid.NewGuid().ToString();
    }

    public Identity(string id)
    {
      this.Id = id;
    }

    public string Id { get; set; }

    public bool Equals(Identity id)
    {
      if (object.ReferenceEquals(this, id)) return true;
      if (object.ReferenceEquals(null, id)) return false;
      return this.Id.Equals(id.Id);
    }

    public override bool Equals(object anotherObject)
    {
      return Equals(anotherObject as Identity);
    }

    public override int GetHashCode()
    {
      return (this.GetType().GetHashCode() * 907) + this.Id.GetHashCode();
    }

    public override string ToString()
    {
      return this.GetType().Name + " [Id=" + Id + "]";
    }
  }

  public interface IIdentity
  {
```

```
  string Id { get; set; }
}
```

So, the ProductState object stands on its own when it comes to persisting the state of the Product. However, the ProductState also holds another collection of entities; that is, the List of ProductBacklogItemState:

```
public class ProductState
{
  [Key]
  public string ProductKey { get; set; }
  ...
  public List<ProductBacklogItemState> BacklogItems { get; set; }
  ...
}
```

This is all well and good because we keep the database mappings really simple. Yet, how do we get a ProductBacklogItemState object, or the entire List collection for that matter, into a format that we can allow clients to consume? The ProductBacklogItemState is an internal implementation details—just a data holder. This points to the need for a few simple converters, which are used by the Product Aggregate root:

```
public class Product
{
  ...
  public ICollection AllBacklogItems()
```

```
  public Collection AllBacklogItems()
  {
    List all =
        State.BacklogItems.ConvertAll(
            new Converter<ProductBacklogItemState, ProductBacklogItem>(
                ProductBacklogItemState.ToProductBacklogItem));

    return new ReadOnlyCollection(all);
  }

  public ProductBacklogItem BacklogItem(BacklogItemId backlogItemId)
  {
    ProductBacklogItemState state =
        State.BacklogItems.FirstOrDefault(
            x => x.BacklogItemKey.Equals(backlogItemId.Id));

    return new ProductBacklogItem(state);
  }
  ...
}
```

Here we convert a collection of ProductBacklogItemState instances to a collection of ProductBacklogItem instances. And when the client requests just one ProductBacklogItem, we convert to one from a single ProductBacklogItemState with the matching identity. The ProductBacklogItemState object must only support a few simple conversion methods:

```
public class ProductBacklogItemState
{
  [Key]
  public string BacklogItemKey { get; set; }
```

```
  public String BacklogItemKey { get; set; }
  ...
  public ProductBacklogItem ToProductBacklogItem()
  {
    return new ProductBacklogItem(this);
  }

  public static ProductBacklogItem ToProductBacklogItem(
        ProductBacklogItemState state)
  {
    return new ProductBacklogItem(state);
  }
  ...
}
```

Should the client ask repeatedly for a collection of ProductBacklogItem instances the Product could cache the collection after the first time it is generated.

In the end our goal is to stay out of the way of Entity Framework and make it super simple to map state objects in and out of the database. I think when you consider the DbContext for this solution you will conclude that we have a really simple approach:

```
public class AgilePMContext : DbContext
{
  public DbSet<ProductState> Products { get; set; }
  public DbSet<ProductBacklogItemState> ProductBacklogItems { get; set; }
```

```
  public DbSet<BacklogItemState> BacklogItems { get; set; }
  public DbSet<TaskState> Tasks { get; set; }
  public DbSet<ReleaseState> Releases { get; set; }
  public DbSet<ScheduledBacklogItemState> ScheduledBacklogItems { get; set; }
  public DbSet<SprintState> Sprints { get; set; }
  public DbSet<CommittedBacklogItemState> CommittedBacklogItems { get; set; }
  ...
}
```

Creating and using a ProductRepository is easy as well:

```
public interface ProductRepository
{
  void Add(Product product);
```

```
  Product ProductOfId(TenantId tenantId, ProductId productId);
}

public class EFProductRepository : ProductRepository
{
  private AgilePMContext context;

  public EFProductRepository(AgilePMContext context)
  {
    this.context = context;
  }

  public void Add(Product product)
  {
    try
    {
      context.Products.Add(product.State);
    }
    catch (Exception e)
    {
      Console.WriteLine("Add() Unexpected: " + e);
    }
  }

  public Product ProductOfId(TenantId tenantId, ProductId productId)
  {
    string key = tenantId.Id + ":" + productId.Id;
    var state = default(ProductState);

    try
    {
      state = (from p in context.Products
               where p.ProductKey == key
               select p).FirstOrDefault();
    }
```

```
    catch (Exception e)
    {
      Console.WriteLine("ProductOfId() Unexpected: " + e);
    }

    if (EqualityComparer<ProductState>.Default.Equals(state, default(ProductState)))
    {
      return null;
    }
    else
    {
      return new Product(state);
    }
  }
}

// Using the repository
using (var context = new AgilePMContext())
{
  ProductRepository productRepository = new EFProductRepository(context);

  var product =
        new Product(
              new ProductId(),
              new ProductOwnerId(),
              "Test",
              "A test product.");

  productRepository.Add(product);

  context.SaveChanges();

  ...
  var foundProduct = productRepository.ProductOfId(product.ProductId);
}
```

Taking this approach will help us to stay focused on what really counts the most, our Core Domain and its Ubiquitous Language.

Vaughn Vernon

Vaughn Vernon is a software developer and architect with more than 30 years of experience in a broad range of business domains. Vaughn is a leading expert in Domain-Driven Design, and a champion of simplicity and reactive systems. He consults and teaches around Domain-Driven Design and reactive software development, helping teams and organizations realize the potential of business-driven and reactive systems as they transition from technology-driven legacy web implementation approaches. As he does so, he puts strong emphasis on embracing simplicity whenever possible. Vaughn is the author of three books: Implementing Domain-Driven Design, Reactive Messaging Patterns with the Actor Model, and Domain-Driven Design Distilled, all published by Addison-Wesley.

More to explore

Using vlingo/PLATFORM On AWS Fargate (https://kalele.io/vlingo-on-aws/)
March 13, 2020

I consider myself a refugee from the old JEE architectures. Threading was handled (naively, for the most part) by the container. Requests

(https://kalele.io/vlingo-on-aws/)

(https://kalele.io/are-you-a-techie/)

Are You A Techie? (https://kalele.io/are-you-a-techie/)

February 29, 2020

Are you a technical person? I am often asked that question. Why? Does the answer matter? I believe most are curious and

Announcing vlingo/PLATFORM Version 1.0.0 General Availability (https://kalele.io/announcing-vlingo-platform-version-1-0-0-general-availability/)

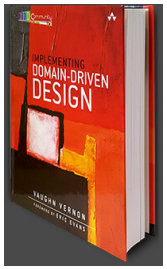
January 10, 2020

Carefree, AZ USA, January 10, 2020 Today Vlingo announced a major milestone, the release of the vlingo/PLATFORM version 1.0.0 GA. General availability

Mission

We aggressively advance software developer skills utilizing DDD and the vlingo/PLATFORM to deliver excellent software solutions. We are committed to balancing the right technology choices with your essential and unique business vision. We champion simplicity, which requires special discipline and determination.

Learn More



(/books/)

Consulting (https://kalele.io/consulting/)

Live Training (https://kalele.io/live-training/)

Online Learning (https://kalele.io/online-learning/)

Learn Now (https://kalele.io/learn-now/)

Blog (https://kalele.io/blog/)

About (https://kalele.io/about/)

Team (https://kalele.io/team/)

News (https://kalele.io/news/)

Contact (https://kalele.io/contact/)

Other Resources

Public Workshops (https://idddworkshop.com/)

Terms & Conditions/License (https://kalele.io/terms-conditions/)

Privacy Policy (https://kalele.io/privacy-policy/)

Email Us (mailto:info@kalele.io)

YouTube Channel (https://www.youtube.com/channel/UCdbDxsXevDLt7EhRbi2KGjg?sub_confirmation=1)

Kalele Retweeted

Wolfgang Werner
@0xcafebabe

Chances are you are stuck at home at 6-7PM CET today, so why not join @VaughnVernon and me talking about vlingo-schemata, @vlingo_io's schema registry. For more details on what to expect see github.com/vlingo/vlingo-… and docs.vlingo.io/vlingo-schemata

Register -> register.gotowebinar.com/register/87476…

vlingo/schemata
docs.vlingo.io

8h