Object Management Stack

- Overview
- Object Ids
- Base Domain Type
- Example Derived Type
- Type Registry Helper

Overview

This is a series of pages that describe a generic object management stack.

The idea being that it will have:

Common Object Properties

All object types will be identified and referenced by a UUIDv7.

The identifier shall be called an ObjID, or just Id, if in EF.

See this page for Object Identification: Object Ids

Object Ids

Every object is uniquely identifiable, by a standardized UUIDv7.

This datatype works across MSSQL, PostgreSQL, SQLite, and Typescript/JavaScript (as string).

In Service and Storage

In API, services, and backend storage, an object's ID is created as an UUIDv7.

And, it is handled and stored as a GUID.

Id Over The Wire

Over the wire, an object id can be passed in Guid form.

But a more useful way is to return it in a form, similar to a MoRef.

This is a two-term string of type and Id:

<objecttype>:<objectid>

The type is a shorthand type, recognizable by a UI.

And, the objectid is a condensed version of the object's Id (its UUIDv7 Id).
This objectid is encoded with the NanoId library, which creates condensed Base64 versions of UUID.

Benefits

Here are the benefits of the design choice, above.

- Resources/Object are identified by UUIDv7.
- A UUIDv7 is time sortable.
- It is an acceptable key type for PostgreSQL, MSSQL, SQLite, and JS libraries.
- Client UIs can mint object Ids, that are acceptable by the backend.

Base Domain Type

Here are notes about the base domain type, its usage, and design choices.

> NOTE: Some of these are restrictions of the C# compiler, that makes it necessary to include some boilerplate in domain objects.

Design Requirements

Here are design requirements for our domain base:

- Derived type instances shall be fully-formed at construction
- Support persistence by EF
- Support persistence by Dapper
- Support persistence by In-Memory backends
- Derived type instances must be reference-able at construction
- Include a protected constructor that accepts an Id
- Include a parameter-less, protected constructor
- Include creation and modified timestamp
- Include a setter for the Modified time
- Properties are publicly accessible, privately set

The above requirements come from these constraints:

Be Fully Formed

We want domain entities to be fully-formed at construction.
This is so that our domain logic doesn't have to save an instance to a database, before it can be used or referenced.

Instead, the domain logic can treat any new instance or copy of one, to be usable in logic, without being first saved to a backend.

Often, we regard this as making our object instances be "alive" or "exists" at construction.

This means that every instance shall have an identity and meaningful timestamps and properties at conception.

What this allows us to do, is to generate instances, remotely, and disconnected from a database.

It allows us to craft domain logic that saves those instances to the backend, as the domain logic sees fit.
Meaning: A disconnected client may create objects and work with them, while there is no connection to the backend.

Support EF Persistence

This technology requires the presence of protected constructors, so that instances can be hydrated from storage, via reflection.

This means that we require a parameter-less, protected constructor in the base type.

AND: This means that the derived type also requires the presence of a parameter-less, protected constructor.

See this for the reasons: C# Compiler Problem

Support Dapper

This ORM technology is more hands-on than EF, in that it doesn't include change tracking.
So, we must include explicit logic to set the creation and modified timestamps of entities.

Similar to EF, Dapper requires the presence of protected constructors, so that instances can be hydrated from storage, via reflection.
This means that we require a parameter-less, protected constructor in the base type.

AND: This means that the derived type also requires the presence of a parameter-less, protected constructor.

See this for the reasons: C# Compiler Problem

Support In-Memory Backend

Using the domain type with an in-memory backend, means that not everything is managed.
Specifically, EF includes change tracking that handles setting creation and modified timestamps for us, when saving instances to a backend.

For example: An in-memory backend doesn't include any change-tracking that can set modified timestamps.
So, our base type does some things, explicitly, in case it is used outside of EF.

Our domain base needs to include needs a setter for its modified timestamp, so any domain logic can update it.

Our domain base needs to set its own creation timestamp at construction.

Reference-able at Construction

Domain entities shall have an identity on construction.

What we mean is, that we don't have to save an instance to a database, for it to have a valid Id.

This means, that an instance can be created, remotely and disconnected from a database, and still be fully usable in domain logic.

Specifically, we can create an entire graph of referenced and referencing entities, without ever touching a database.
This allows our domain model to function on a mobile device or browser that has lost network connectivity to its backend.

Protected Constructor Accepts Id

Our base type mandatorily accepts its identity at construction.

This is not a constraint of EF or Dapper, as an ORM.
It is specified, here, as a constraint, for how we declare identity in the domain.

To implement this, the base type includes a protected constructor that accepts the identity value.

> NOTE: Identity is not generated by the base type.
> BUT: Identity is applied by the base type, as no derived type has access to change it.

Derived types shall pass an identity value to the base constructor.

How they accomplish this, can be based on their intent:

- For domain types that can be created remotely or disconnected, these types may accept an identity in their public constructor.
  And, that identity value gets passed to the base type, like this constructor:

```
public DerivedType(Guid id, string name) : base(id)
```

- For regular domain types, their identity can be created in the constructor, like this:

```
public DerivedType(string name) : base(Guid.NewGuid())
```

In either method, the identity shall be set at construction.

> NOTE: This has no bearing on how an instance's identity is set during hydration, as that is done, via reflection.

Parameterless, Protected Constructor

As mentioned, earlier, our base type needs to support persistence by EF and Dapper.

To support this, our domain base needs to include a parameterless, protected constructor.
This is required by EF and Dapper, for proper hydration from storage, via reflection.

As well: This means that the derived type also requires the presence of a parameter-less, protected constructor.

See this for the reasons: C# Compiler Problem

Creation Timestamp

Our domain base needs to include audit timestamping of when created and modified.

Creation time shall be set at construction.

If not set at construction, the SaveChanges() method in EF will set the creation time if NULL.

Modified Timestamp

Our domain base needs to include audit timestamping of when created and modified.

Modified time shall be NULL at construction.

Modified time can be set through a public setter method.
This allows domain logic to update the timestamp as the instance mutates.

Public Getter

Every derived type shall include public getters for each property it contains.

This allows domain and service logic to inspect, or get, the state of an entity.

Protected Setter

Every derived type shall include protected setters for each property it contains.

This prevents the direct mutation of entity state.

And, it allows the entity to enforce validation and business rules for any changes to state.

> NOTE: It is important to note that EF and Dapper do not use the logic in any setter, when hydrating instances from storage.
> These ORMs apply reflection, to directly set public properties.

A public setter method shall do the following:

- Receive a value of the same type as the public property
- Determine if the property will be changed by the given value
  If the same, the setter simply returns.
- Validate the new value.
  If it fails validation checks, throw a domain exception.
- Update the property with the new value.
- Update the Modified timestamp.

C# Compiler Problem

> NOTE: This problem applies to EF and Dapper backends.
> But, is a general limitation of the C# compiler.

When the C# compiler encounters a type that derives from a base, it inspects the derived type for any constructors.

If the derived type includes a constructor, regardless of its visibility (public, protected, private), any constructors that are in the base class, will be hidden and excluded, unless explicitly called.

What makes this a problem is: EF and Dapper both require a parameter-less, protected constructor, in order to properly hydrate instances from storage.

And, EF and Dapper will have problems if they cannot identify a parameter-less, protected constructor to call.

So, our domain base type requires a parameter-less, protected constructor that effectively does nothing.

Example Base Type

Fulfilling the above design requirements and constraints, gives us a domain base that looks like the following:

```
    /// <summary>
    /// Base type for all domain models.
    /// Derive from this, to have a persistable domain type.
    /// This is purposely made abstract, so it's well known that the base type will not be persisted.
    /// EF will not try to map to it, directly, will not create a table for it, nor persist it directly.
    /// </summary>
    abstract public class DomainBase_v1
    {
        public Guid Id { get; protected set; }

        /// <summary>
        /// Is nullable, for DB compatibility and to allow for detached instances.
        /// Will be set on construction, and when saved to a backend as new object.
        /// </summary>
        public DateTimeOffset CreationDateUTC { get; protected set; }

        /// <summary>
        /// Is nullable, for DB compatibility and to allow for detached instances.
        /// Will be set when the instance is modified, and saved to the backend as an updated instance.
        /// </summary>
        public DateTimeOffset? ModifiedDateUTC { get; protected set; }

        /// <summary>
        /// Required for a derived object type to chain to.
        /// Indirectly, this Protected parameterless constructor is needed for EF + Dapper hydration.
        /// NOTE:   This constructor is required on the base type, so that the derived type's constructor has
something to chain to.
        /// NOTE:   If your derived type includes any constructor, at all, then this parameter-less constructor will be
hidden and not visible to EF or Dapper.
        ///         So, you will need to include a parameter-less, protected constructor in your derived type, to allow
this one to be visible, IF your derived type has at least one constructor of its own.
        /// </summary>
        protected DomainBase_v1() { }

        /// <summary>
        /// Optional domain constructor.
        /// NOTE: This base accepts an identity, for derived types that need to work disconnected, and be considerd
as "live" domain objects at construction.
        /// NOTE: If your derived type doesn't need to define the identity, then simply pass a new Id to the base, like
this constructor signature:
```

```
        ///     public DerivedType(string name) : base(Guid.NewGuid())
        /// NOTE: If your derived type needs to accept an indentity at construction, then pass the given identity to the base, like this constructor signature:
        ///     public DerivedType(Guid id, string name) : base(id)
        /// </summary>
        /// <param name="id"></param>
        protected DomainBase_v1(Guid id)
        {
            // Ensure the identity is not empty...
            if(id == Guid.Empty)
                throw new ArgumentException("Id cannot be empty.", nameof(id));

            // Accept the identity...
            Id = id;

            // NOTE: We set the creation time, at construction, here, so that any in-memory persistence can have a timestamp to work with and use.
            // Normal persistence (to MSSQL, SQLite, Postgres) will set this in the SaveChanges() override, for added instances.
            CreationDateUTC = DateTimeOffset.UtcNow;

            // NOTE: We set the modified time to match creation, on construction.
            // This allows us to simplify queries that filter on modified times.
            // If our domain requires explicit change tracking, we will impose a version identifier, as our indicator of change.
            this.ModifiedDateUTC = this.CreationDateUTC;
        }

        /// <summary>
        /// For EF, this will be called, automatically, on SaveChanges().
        /// For non-EF, call this method when your logic updates the instance.
        /// NOTE: This base domain type assumes the modified time will be UTC.
        /// </summary>
        public void TouchModified() => ModifiedDateUTC = DateTimeOffset.UtcNow;
    }
```

The base type includes a few design details, explained here.

Abstract Type

The base type is abstract.

This means that the ORM will not attempt to create instances of the base.

Parameter-less Protected Constructor

See the above design constraints for an explanation of the parameterless, protected constructor.

Simply: This is required in the base type, to allow constructor chaining to work.
This is needed in the base type, so that constructors of derived types can chain to it.
This is a compiler requirement, not an EF or Dapper requirement.

But indirectly, EF and Dapper require a parameter-less, protected constructor in the derived type.
And, that needs something to chain to, on the base, making this important.

Protected Constructor with Id Parameter

See the above design constraints for an explanation of the protected constructor with Id.

The base type includes a protected constructor that accepts an Id.

This is NOT an EF or Dapper requirement.

It is present, so that derived types can supply an Id when needed.
This would be used where domain objects can be created, remotely or disconnected from the backend, where the instance must be fully alive and reference-able without having, yet, been saved to a backend.

Id Property

This is our primary key of the object in storage.

It is how we find the instances for retrieval.

> NOTE: The Id is ALWAYS accepted at construction.
> Meaning: The constructor of your derived type, needs to pass the id it receives, or create an id value.

The identity (Id) is given to the base type for this reason:

- So that instances can be considered "alive" and "exist", and be fully created (having an Id)

or when an instance is saved to the backend.

It is also set in the instance, when hydrated from storage, via reflection.

Creation Timestamp

This property is defined on the base type, so that every derived type has tracking of when instances were created.

It's an interesting design choice, of when to set this:

- We could set it at construction, if we consider the instance "live" for domain consistency.
- We could set it when saved to the backend, to indicate when the instance was considered usable by any read-model or other logic.

We take a hybrid approach, that satisfies several use cases.
And in doing so, we follow these rules, for the creation time property:

- We set the creation time at instance construction.
  This ensures that an instance is "alive" or "exists" before commit.
  And, it allows us to track audit metadata.
  This also allows for object types to function with an In-Memory backend, that has no logic to set creation timestamps during persistence.
- We also set the creation time, when a new instance is saved to the backend.
  This is done by an override of SaveChanges(), in our DbContext type.
  This is action is only done if the object's creation time is NULL.
  And, is done to enforce correctness at persistence time.

Modified Timestamp

This property is defined on the base type, so that every derived type has tracking of when instances were updated.

It's an interesting design choice, of when to set this:

- We could set it by a domain-logic Setter.
- We could set it when the updated instance is saved to the backend, to indicate when the update was saved.

We take a hybrid approach, that satisfies a few use cases.
And in doing so, we follow these rules, for the modified time property:

- We leave the modified time as NULL at instance construction.
- We include a setter, called TouchModified(), to update the modified time when the object is updated.
  This can be called by any domain logic that mutates the instance.
  And, it allows for the domain logic to fully define when the change occurs, as an "alive" instance.
  And, it allows us to fully track audit metadata.
  This also allows for object types to function with an In-Memory backend, that has no logic

to set modified timestamps during persistence.
- We also set the modified time, when an updated instance is saved to the backend.
  This is done by an override of SaveChanges(), in our DbContext type.
  This is action is done if the change tracking logic identifies the instance as updated.

Example Derived Type

See this page for examples of domain types that derive from the above base: Example Derived Type

Example Derived Type

Here's a working example of a domain type that derives from the base type in this page: Base Domain Type

```
    /// <summary>
    /// This is a sample domain type that uses our base type.
    /// It is only for testing.
    /// For this type to exist, it requires:
    ///     A model builder definition.
    ///     A table definition that can be generated in the backend, for storing instances.
    /// </summary>
    public class DOWidget : DomainBase_v1
    {
        #region public Properties

        public string Name { get; protected set; }

        #endregion



        #region ctor

        /// <summary>
        /// EF & Dapper require a parameterless constructor for hydration, in any type that includes its own constructor.
        /// MEANING: Because of constructor-chaining:
        ///     If your derived class defines its own constructors (even just one, like the argument-based constructor that Widget includes),
        ///     then the compiler hides the base's parameter-less constructor.
        ///     This means that you need to make it a practice, to include a parameter-less protected constructor, like this, as boilerplate in all derived types.
        ///     This ensures that EF and Dapper have access to the protected parameter-less constructor.
        /// </summary>
        protected DOWidget() { }

        /// <summary>
```

```
        /// Domain constructor, used by code, to create instances.
        /// NOTE: The base type applies identity, but the derived type creates it.
        /// </summary>
        /// <param name="id"></param>
        /// <param name="name"></param>
        public DOWidget(string name) : base(Guid.NewGuid())
        {
            // Set any received properties.
            Name = name;
        }

        /// <summary>
        /// This variant can accept an Id at construction.
        /// NOTE:   Whether or not you pass in an Id at construction has no bearing on how EF or Dapper handles it.
        ///         This choice is simply based on how your domain logic creates new instances.
        ///         If you create instances, remotely (read as disconnected), include a constructor that can accept an
Id.
        /// </summary>
        /// <param name="id"></param>
        /// <param name="name"></param>
        public DOWidget(Guid id, string name) : base(id)
        {
            if(!Name_IsValid(name))
                throw new ArgumentException("Name cannot be empty.", nameof(name));

            // Accept given properties...



            Name = name;
        }

        #endregion



        #region Setters

        /// <summary>
        /// Public setter for the Name property.
        /// </summary>
        /// <param name="val"></param>
```

```
    /// <exception cref="ArgumentException"></exception>
    public void Rename(string val)
    {
        // See if a change to deal with...
        if(this.Name != val)
        {
            // No change.
            return;
        }

        // Validate the input...
        if(!Name_IsValid(val))
            throw new ArgumentException("Name cannot be empty.", nameof(val));

        // Accept the new name...
        Name = val.Trim();

        // Update modified date...
        TouchModified();
    }

    #endregion


    #region Validation Methods

    static public bool Name_IsValid(string val)
    {
        if (string.IsNullOrWhiteSpace(val))
            return false;

        var reg = new Regex(@"^[a-zA-Z0-9_.-]+$");
        if (reg.IsMatch(val))
            return true;
        else
            return false;
    }

    #endregion
}
```

The above domain type, has the following behaviors and properties:

- All properties are public get, and protected set.
  This allows domain logic to get state, while not being able to directly touch it.
- It includes the parameter-less, protected constructor.
  This is required by EF and Dapper, for instance hydration from storage.
- It includes a public constructor that accepts enough parameters, to fully-form an instance.
  This allows any instance to be well-formed, and fully-formed, at all times.
- The public constructor, purposely, does NOT accept all properties of the base.
  Specifically, it is missing the creation and modified timestamps.
  Aside from it not making sense to include these.
  Not including these in the public constructor, ensures that EF and Dapper will NEVER use the public constructor to materialize instances.
  So, the public constructor can safely contain validation logic, that will never be traversed by EF or Dapper.
  This ensures us that our validation logic will never trip up our ORM.
- Our public Setter, short-circuits if no change.
- Our public Setter validates any candidate values.
- Our public Setter throws an exception on validation failure.
- Our public Setter updates the Modified timestamp, if an update occurs.
- Our property validation logic is consolidated, so the constructor and Setter methods can share it.

Type Registry Helper

TypeRegistryHelper