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.

Revision #4
Created 5 November 2025 04:57:32 by glwhite
Updated 11 November 2025 11:42:45 by glwhite