4.1 (latest) English

Search in docume

Contributors

Edit Last edit: 8/8/2020

Share on :

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

In this document

Entities

Entities are one of the core concepts of DDD (Domain Driven Design). Eric Evans describe it as "*An object that is not fundamentally defined by its attributes, but rather by a thread of continuity and identity*".

An entity is generally mapped to a table in a relational database.

Entity Class

Entities are derived from the `Entity<TKey>` class as shown below:

```
public class Book : Entity<Guid>
{
    public string Name { get; set; }

    public float Price { get; set; }
}
```

> If you do not want to derive your entity from the base `Entity<TKey>` class, you can directly implement `IEntity<TKey>` interface.

`Entity<TKey>` class just defines an `Id` property with the given primary **key type**, which is `Guid` in the example above. It can be other types like `string` , `int` , `long` or whatever you need.

Entities with GUID Keys

If your entity's Id type is `Guid` , there are some good practices to implement:

- Create a constructor that gets the Id as a parameter and passes to the base class.
  - If you don't set a GUID Id, **ABP Framework sets it on save**, but it is good to have a valid Id on the entity even before saving it to the database.
- If you create an entity with a constructor that takes parameters, also create a `private` or `protected` empty constructor. This is used while your database provider reads your entity from the database (on deserialization).
- Don't use the `Guid.NewGuid()` to set the Id! **Use the `IGuidGenerator` service** while passing the Id from the code that creates the entity. `IGuidGenerator` optimized to generate sequential GUIDs, which is critical for clustered indexes in the relational databases.

An example entity:

4.1 (latest) English

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

Share on :

In this document

```
public class Book : Entity<Guid>
{
    public string Name { get; set; }

    public float Price { get; set; }

    protected Book()
    {

    }

    public Book(Guid id)
     : base(id)
    {

    }
}
```

Example usage in an application service:

```
public class BookAppService : ApplicationService, IBook
{
    private readonly IRepository<Book> _bookRepository;

    public BookAppService(IRepository<Book> bookReposit
    {
        _bookRepository = bookRepository;
    }

    public async Task CreateAsync(CreateBookDto input)
    {
        await _bookRepository.InsertAsync(
            new Book(GuidGenerator.Create())
            {
                Name = input.Name,
                Price = input.Price
            }
        );
    }
}
```

- `BookAppService` injects the default repository for the book entity and uses its `InsertAsync` method to insert a `Book` to the database.
- `GuidGenerator` is type of `IGuidGenerator` which is a property defined in the `ApplicationService` base class. ABP defines such frequently used base properties as pre-injected for you, so you don't need to manually inject them.
- If you want to follow the DDD best practices, see the *Aggregate Example* section below.

Entities with Composite Keys

Some entities may need to have **composite keys**. In that case, you can derive your entity from the non-generic `Entity` class. Example:

4.1 (latest) English

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

Share on :

In this document

```
public class UserRole : Entity
{
    public Guid UserId { get; set; }

    public Guid RoleId { get; set; }

    public DateTime CreationTime { get; set; }

    public UserRole()
    {

    }

    public override object[] GetKeys()
    {
        return new object[] { UserId, RoleId };
    }
}
```

For the example above, the composite key is composed of `UserId` and `RoleId`. For a relational database, it is the composite primary key of the related table. Entities with composite keys should implement the `GetKeys()` method as shown above.

> Notice that you also need to define keys of the entity in your **object-relational mapping** (ORM) configuration. See the Entity Framework Core integration document for example.

> Also note that Entities with Composite Primary Keys cannot utilize the `IRepository<TEntity, TKey>` interface since it requires a single Id property. However, you can always use `IRepository<TEntity>`. See repositories documentation for more.

AggregateRoot Class

"*Aggregate is a pattern in Domain-Driven Design. A DDD aggregate is a cluster of domain objects that can be treated as a single unit. An example may be an order and its line-items, these will be separate objects, but it's useful to treat the order (together with its line items) as a single aggregate.*" (see the full description)

`AggregateRoot<TKey>` class extends the `Entity<TKey>` class. So, it also has an `Id` property by default.

> Notice that ABP creates default repositories only for aggregate roots by default. However, it's possible to include all entities. See the repositories documentation for more.

ABP does not force you to use aggregate roots, you can in fact use the `Entity` class as defined before. However, if you want to implement the Domain Driven Design and want to create aggregate root classes, there are some best practices you may want to consider:

- An aggregate root is responsible to preserve it's own integrity. This is also true for all entities, but aggregate root has responsibility for it's sub entities too. So, the aggregate root must always be in a valid state.

- An aggregate root can be referenced by it's Id. Do not reference it by it's navigation property.
- An aggregate root is treated as a single unit. It's retrieved and updated as a single unit. It's generally considered as a transaction boundary.
- Work with sub-entities over the aggregate root- do not modify them independently.

See the entity design best practice guide if you want to implement DDD in your application.

Aggregate Example

This is a full sample of an aggregate root with a related sub-entity collection:

Share on :

In this document

4.1 (latest) English

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

```
public class Order : AggregateRoot<Guid>
{
    public virtual string ReferenceNo { get; protected

    public virtual int TotalItemCount { get; protected

    public virtual DateTime CreationTime { get; protect

    public virtual List<OrderLine> OrderLines { get; pr

    protected Order()
    {

    }

    public Order(Guid id, string referenceNo)
    {
        Check.NotNull(referenceNo, nameof(referenceNo))

        Id = id;
        ReferenceNo = referenceNo;

        OrderLines = new List<OrderLine>();
    }

    public void AddProduct(Guid productId, int count)
    {
        if (count <= 0)
        {
            throw new ArgumentException(
                "You can not add zero or negative count
                nameof(count)
            );
        }

        var existingLine = OrderLines.FirstOrDefault(ol

        if (existingLine == null)
        {
            OrderLines.Add(new OrderLine(this.Id, produ
        }
        else
        {
            existingLine.ChangeCount(existingLine.Count
        }

        TotalItemCount += count;
    }
}

public class OrderLine : Entity
{
    public virtual Guid OrderId { get; protected set; }

    public virtual Guid ProductId { get; protected set;

    public virtual int Count { get; protected set; }

    protected OrderLine()
    {
```

Share on :

In this document

Share on :

In this document

```
        }

        internal OrderLine(Guid orderId, Guid productId, in
        {
            OrderId = orderId;
            ProductId = productId;
            Count = count;
        }

        internal void ChangeCount(int newCount)
        {
            Count = newCount;
        }

        public override object[] GetKeys()
        {
            return new Object[] {OrderId, ProductId};
        }
    }
```

> If you do not want to derive your aggregate root from the base `AggregateRoot<TKey>` class, you can directly implement the `IAggregateRoot<TKey>` interface.

`Order` is an **aggregate root** with `Guid` type `Id` property. It has a collection of `OrderLine` entities. `OrderLine` is another entity with a composite primary key (`OrderId` and `ProductId`).

While this example may not implement all the best practices of an aggregate root, it still follows some good practices:

- `Order` has a public constructor that takes **minimal requirements** to construct an `Order` instance. So, it's not possible to create an order without an id and reference number. The **protected/private** constructor is only necessary to **deserialize** the object while reading from a data source.
- `OrderLine` constructor is internal, so it is only allowed to be created by the domain layer. It's used inside of the `Order.AddProduct` method.
- `Order.AddProduct` implements the business rule to add a product to an order.
- All properties have `protected` setters. This is to prevent the entity from arbitrary changes from outside of the entity. For exmple, it would be dangerous to set `TotalItemCount` without adding a new product to the order. It's value is maintained by the `AddProduct` method.

ABP Framework does not force you to apply any DDD rule or patterns. However, it tries to make it possible and easier when you do want to apply them. The documentation also follows the same principle.

Aggregate Roots with Composite Keys

While it's not common (and not suggested) for aggregate roots, it is in fact possible to define composite keys in the same way as defined for the mentioned entities above. Use non-generic `AggregateRoot` base class in that case.

BasicAggregateRoot Class

4.1 (latest) English

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

Share on :

In this document

`AggregateRoot` class implements the `IHasExtraProperties` and `IHasConcurrencyStamp` interfaces which brings two properties to the derived class. `IHasExtraProperties` makes the entity extensible (see the *Extra Properties* section below) and `IHasConcurrencyStamp` adds a `ConcurrencyStamp` property that is managed by the ABP Framework to implement the optimistic concurrency. In most cases, these are wanted features for aggregate roots.

However, if you don't need these features, you can inherit from the `BasicAggregateRoot<TKey>` (or `BasicAggregateRoot`) for your aggregate root.

Base Classes & Interfaces for Audit Properties

There are some properties like `CreationTime`, `CreatorId`, `LastModificationTime` ... which are very common in all applications. ABP Framework provides some interfaces and base classes to **standardize** these properties and also **sets their values automatically**.

Auditing Interfaces

There are a lot of auditing interfaces, so you can implement the one that you need.

> While you can manually implement these interfaces, you can use **the base classes** defined in the next section to simplify it.

- `IHasCreationTime` defines the following properties:
  - `CreationTime`
- `IMayHaveCreator` defines the following properties:
  - `CreatorId`
- `ICreationAuditedObject` inherits from the `IHasCreationTime` and the `IMayHaveCreator`, so it defines the following properties:
  - `CreationTime`
  - `CreatorId`
- `IHasModificationTime` defines the following properties:
  - `LastModificationTime`
- `IModificationAuditedObject` extends the `IHasModificationTime` and adds the `LastModifierId` property. So, it defines the following properties:
  - `LastModificationTime`
  - `LastModifierId`
- `IAuditedObject` extends the `ICreationAuditedObject` and the `IModificationAuditedObject`, so it defines the following properties:
  - `CreationTime`
  - `CreatorId`
  - `LastModificationTime`
  - `LastModifierId`
- `ISoftDelete` (see the data filtering document) defines the following properties:
  - `IsDeleted`
- `IHasDeletionTime` extends the `ISoftDelete` and adds the `DeletionTime` property. So, it defines the following properties:
  - `IsDeleted`
  - `DeletionTime`
- `IDeletionAuditedObject` extends the `IHasDeletionTime` and adds the `DeleterId` property. So, it defines the following properties:
  - `IsDeleted`
  - `DeletionTime`

4.1 (latest) English

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

Share on :

In this document

  - `DeleterId`
- `IFullAuditedObject` inherits from the `IAuditedObject` and the `IDeletionAuditedObject`, so it defines the following properties:
  - `CreationTime`
  - `CreatorId`
  - `LastModificationTime`
  - `LastModifierId`
  - `IsDeleted`
  - `DeletionTime`
  - `DeleterId`

Once you implement any of the interfaces, or derive from a class defined in the next section, ABP Framework automatically manages these properties wherever possible.

> Implementing `ISoftDelete`, `IDeletionAuditedObject` or `IFullAuditedObject` makes your entity **soft-delete**. See the data filtering document to learn about the soft-delete pattern.

Auditing Base Classes

While you can manually implement any of the interfaces defined above, it is suggested to inherit from the base classes defined here:

- `CreationAuditedEntity<TKey>` and `CreationAuditedAggregateRoot<TKey>` implement the `ICreationAuditedObject` interface.
- `AuditedEntity<TKey>` and `AuditedAggregateRoot<TKey>` implement the `IAuditedObject` interface.
- `FullAuditedEntity<TKey>` and `FullAuditedAggregateRoot<TKey>` implement the `IFullAuditedObject` interface.

All these base classes also have non-generic versions to take `AuditedEntity` and `FullAuditedAggregateRoot` to support the composite primary keys.

All these base classes also have `...WithUser` pairs, like `FullAuditedAggregateRootWithUser<TUser>` and `FullAuditedAggregateRootWithUser<TKey, TUser>`. This makes possible to add a navigation property to your user entity. However, it is not a good practice to add navigation properties between aggregate roots, so this usage is not suggested (unless you are using an ORM, like EF Core, that well supports this scenario and you really need it - otherwise remember that this approach doesn't work for NoSQL databases like MongoDB where you must truly implement the aggregate pattern). Also, if you add navigation properties to the AppUser class that comes with the startup template, consider to handle (ignore/map) it on the migration dbcontext (see the EF Core migration document).

Extra Properties

ABP defines the `IHasExtraProperties` interface that can be implemented by an entity to be able to dynamically set and get properties for the entity. `AggregateRoot` base class already implements the `IHasExtraProperties` interface. If you've derived from this class (or one of the related audit class defined above), you can directly use the API.

GetProperty & SetProperty Extension Methods

These extension methods are the recommended way to get and set data for an entity. Example:

Share on :

In this document

```
public class ExtraPropertiesDemoService : ITransientDep
{
    private readonly IIdentityUserRepository _identityU

    public ExtraPropertiesDemoService(IIdentityUserRepo
    {
        _identityUserRepository = identityUserRepositor
    }

    public async Task SetTitle(Guid userId, string titl
    {
        var user = await _identityUserRepository.GetAsy

        //SET A PROPERTY
        user.SetProperty("Title", title);

        await _identityUserRepository.UpdateAsync(user)
    }

    public async Task<string> GetTitle(Guid userId)
    {
        var user = await _identityUserRepository.GetAsy

        //GET A PROPERTY
        return user.GetProperty<string>("Title");
    }
}
```

- Property's **value is object** and can be any type of object (string, int, bool... etc).
- `GetProperty` returns `null` if given property was not set before.
- You can store more than one property at the same time by using different **property names** (like `Title` here).

It would be a good practice to **define a constant** for the property name to prevent typo errors. It would be even a better practice to **define extension methods** to take the advantage of the intellisense. Example:

```
public static class IdentityUserExtensions
{
    private const string TitlePropertyName = "Title";

    public static void SetTitle(this IdentityUser user,
    {
        user.SetProperty(TitlePropertyName, title);
    }

    public static string GetTitle(this IdentityUser use
    {
        return user.GetProperty<string>(TitlePropertyNa
    }
}
```

Then you can directly use `user.SetTitle("...")` and `user.GetTitle()` for an `IdentityUser` object.

4.1 (latest) English

Filter topics

- Migrating from the ASP.NET Boilerplate
- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - Overall
    - Domain Layer
      - Entities & Aggregate Roots
      - Value Objects
      - Repositories
      - Domain Services
      - Specifications
    - Application Layer
      - Application Services
      - Data Transfer Objects
      - Unit Of Work
    - Guide: Implementing DDD
  - Multi Tenancy
  - Microservices
- API
- User Interface
- Data Access
- Real Time
- Testing
- Samples
- Application Modules
- Release Information
- Reference
- Contribution Guide

HasProperty & RemoveProperty Extension Methods

- `HasProperty` is used to check if the object has a property set before.
- `RemoveProperty` is used to remove a property from the object. You can use this instead of setting a `null` value.

Share on :

In this document

How it is Implemented?

`IHasExtraProperties` interface requires to define a `Dictionary<string, object>` property, named `ExtraProperties`, for the implemented class.

So, you can directly use the `ExtraProperties` property to use the dictionary API, if you like. However, `SetProperty` and `GetProperty` methods are the recommended ways since they also check for `null`s.

How is it Stored?

The way to store this dictionary in the database depends on the database provider you're using.

- For Entity Framework Core, here are two type of configurations;
  - By default, it is stored in a single `ExtraProperties` field as a `JSON` string (that means all extra properties stored in a single database table field). Serializing to `JSON` and deserializing from the `JSON` are automatically done by the ABP Framework using the value conversions system of the EF Core.
  - If you want, you can use the `ObjectExtensionManager` to define a separate table field for a desired extra property. Properties those are not configured through the `ObjectExtensionManager` will continue to use a single `JSON` field as described above. This feature is especially useful when you are using a pre-built application module and want to extend its entities. See the EF Core integration document to learn how to use the `ObjectExtensionManager`.
- For MongoDB, it is stored as a **regular field**, since MongoDB naturally supports this kind of extra elements system.

Discussion for the Extra Properties

Extra Properties system is especially useful if you are using a **re-usable module** that defines an entity inside and you want to get/set some data related to this entity in an easy way.

You typically **don't need** to use this system for your own entities, because it has the following drawbacks:

- It is **not fully type safe** since it works with strings as property names.
- It is **not easy to auto map** these properties from/to other objects.

Extra Properties Behind Entities

`IHasExtraProperties` is not restricted to be used with entities. You can implement this interface for any kind of class and use the `GetProperty`, `SetProperty` and other related methods.

See Also

- Best practice guide to design the entities

4.1 (latest) English

Filter topics

→ Migrating from the ASP.NET Boilerplate

- Fundamentals
- Infrastructure
- Architecture
  - Modularity
  - Domain Driven Design
    - → Overall
    - Domain Layer
      - → Entities & Aggregate Roots
      - → Value Objects
      - → Repositories
      - → Domain Services
      - → Specifications
    - Application Layer
      - → Application Services
      - → Data Transfer Objects
      - → Unit Of Work
    - → Guide: Implementing DDD
  - → Multi Tenancy
  - → Microservices
- API
- User Interface
- Data Access
- Real Time
- → Testing
- Samples
- Application Modules
- Release Information
- Reference
- → Contribution Guide

Share on :

In this document