ASP.NET Boilerplate

Articles Tutorials

Filter...

latest (v7.1)

ABP Framework

Overall

- Introduction
- Tutorials & Articles
- NLayer Architecture
- Module System
- Startup Configuration
- Multi-Tenancy
- OWIN Integration
- Debugging
- API Reference

Common Structures

- Dependency Injection
- Session
- Caching
- Logging
- Setting Management
- Timing
- Object To Object Mapping (and AutoMapper Integration)
- Email Sending (and MailKit Integration)

Domain Layer

- Entities
- Multi-Lingual Entities
- Value Objects
- Repositories
- Domain Services
- Specifications
- Unit Of Work
- Domain Events (EventBus)
- Data Filters
- Dynamic Parameter System
- Object Comparators

Application Layer

- Application Services
- Data Transfer Objects
- Validating Data Transfer Objects

In this document

Edit on GitHub

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*".

Essentially, entities have Id's and are stored in a database. An entity is generally mapped to a table in a relational database.

Entity Class

In ASP.NET Boilerplate, Entities are derived from the **Entity** class. See the example below:

Copy

```
public class Person : Entity
{
    public virtual string Name { get; set; }

    public virtual DateTime CreationTime { get; set; }

    public Person()
    {
        CreationTime = DateTime.Now;
    }
}
```

The **Person** class is defined as an entity. It has two properties as well as the **Id** property defined in the Entity base class. The **Id** is the **primary key** of the Entity. The name of the primary keys for all Entities are the same, it is **Id**.

The type of the Id (primary key) can be changed. It is **int** (Int32) by default. If you want to define another type as Id, you should explicitly declare it as shown below:

Copy

```
public class Person : Entity<long>
{
    public virtual string Name { get; set; }

    public virtual DateTime CreationTime { get; set; }

    public Person()
    {
        CreationTime = DateTime.Now;
    }
}
```

You can set it as string, Guid or something else.

Entity class overrides the **equality** operator (==) to easily check if two entities are equal (their Id is equal). It also defines the **IsTransient()** method to check if it has an Id or not.

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.*" (Martin Fowler - see the full description)

While ABP does not enforce you to use aggregates, you may want to create aggregates and aggregate roots in your application. ABP defines the **AggregateRoot** class that extends an Entity to create aggregate root entities for an aggregate.

Domain Events

AggregateRoot defines the **DomainEvents** collection to generate domain events by the aggregate root class. These events are automatically triggered just before the current unit of work is completed. In fact, any entity can generate domain events by implementing the **IGeneratesDomainEvents** interface, but it's common (a best practice) to generate domain events in aggregate roots. That's why it's the default for the AggregateRoot but not for the Entity class.

Conventional Interfaces

In many applications, similar entity properties (and database table fields) are used, like CreationTime, which indicates when an entity was created. ASP.NET Boilerplate provides some useful interfaces to make these common properties explicit and expressive. This provides a way of coding common code for Entities which implement these interfaces.

Auditing

IHasCreationTime makes it possible to use a common property for the '**creation time**' information of an entity. When this interface is implemented, ASP.NET Boilerplate automatically sets the CreationTime to the **current time** when an Entity is inserted into the database.

Copy

```
public interface IHasCreationTime
{
    DateTime CreationTime { get; set; }
}
```

The Person class can be re-written as shown below by implementing the IHasCreationTime interface:

Copy

```
public class Person : Entity<long>, IHasCreationTime
{
    public virtual string Name { get; set; }

    public virtual DateTime CreationTime { get; set; }

    public Person()
    {
        CreationTime = DateTime.Now;
    }
}
```

ICreationAudited extends IHasCreationTime by adding **CreatorUserId**:

Copy

```
public interface ICreationAudited : IHasCreationTime
{
    long? CreatorUserId { get; set; }
}
```

ASP.NET Boilerplate automatically sets the CreatorUserId property to the **current user's id** when saving a new entity. You can also easily implement ICreationAudited by deriving your entity from the **CreationAuditedEntity** class. It also has a generic version for different types of Id properties.

There are also similar interfaces for modifications:

Copy

```
public interface IHasModificationTime
{
    DateTime? LastModificationTime { get; set; }
}

public interface IModificationAudited : IHasModificationTime
{
    long? LastModifierUserId { get; set; }
}
```

ASP.NET Boilerplate automatically sets these properties when updating an entity. You just have to define them for your entity.

If you want to implement all of the audit properties, you can directly implement the **IAudited** interface:

Copy

```
public interface IAudited : ICreationAudited, IModificationAudited
{

}
```

As a shortcut, you can derive from the **AuditedEntity** class instead of directly implementing **IAudited**. The AuditedEntity class also has a generic version for different types of Id properties.

Note: ASP.NET Boilerplate gets the current user's Id from ABP Session.

Soft Delete

Soft delete is a commonly used pattern to mark an Entity as deleted instead of actually deleting it from database. For instance, you may not want to hard delete a User from the database since it has many relations to other tables. The **ISoftDelete** interface is used for this purpose:

Copy

```
public interface ISoftDelete
{
    bool IsDeleted { get; set; }
}
```

ASP.NET Boilerplate implements the soft delete pattern out-of-the-box. When a soft-delete entity is being deleted, ASP.NET Boilerplate detects this, prevents deleting, sets IsDeleted as true, and then updates the entity in the database. It also does not retrieve (select) soft deleted entities from the database by automatically filtering them.

If you use soft delete, you may also want to store information of when an entity was deleted and who deleted it. You can implement the **IDeletionAudited** interface, shown below:

Copy

```
public interface IDeletionAudited : ISoftDelete
{
    long? DeleterUserId { get; set; }

    DateTime? DeletionTime { get; set; }
}
```

As you've probably noticed, IDeletionAudited extends ISoftDelete. ASP.NET Boilerplate automatically sets these properties when an entity is deleted.

If you want to implement all the audit interfaces (creation, modification and deletion) for an entity, you can directly implement **IFullAudited** since it inherits from the others:

Copy

```
public interface IFullAudited : IAudited, IDeletionAudited
{

}
```

As a shortcut, you can derive your entity from the **FullAuditedEntity** class that implements them all.

- NOTE 1: All audit interfaces and classes have a generic version for defining the navigation property to your **User** entity (like ICreationAudited<TUser> and FullAuditedEntity<TPrimaryKey, TUser>).
- NOTE 2: All of them also have an **AggregateRoot** version, like AuditedAggregateRoot.

In some cases, soft-delete entities may be requested to be permanently deleted. In those cases, **IRepository.HardDelete** extension method can be used. This method is currently implemented for EntityFramework 6.x and Entity Framework Core.

If you wish to undelete a soft-deleted entity, you can query the entity by disabling SoftDelete filter then use **Abp.Domain.Entities.EntityExtensions.Undelete(entity)**.

Disable Auditing Fields

In some cases, you might want to programmatically create, update or delete some entities and don't want ASP.NET Boilerplate to automatically set CreatorUserId, LastModifierUserId or DeleterUserId. You can easily disable or enable automatic setting of those fields using the DisableAuditing or EnableAuditing methods on the active unit of work.

Copy

```
using (_unitOfWorkManager.Current.DisableAuditing(AbpAuiditing.CreatorUserId))
{
    // CreatorUserId will not be set by ASP.NET Boilerplate automatically
}

using (_unitOfWorkManager.Current.EnableAuditing(AbpAuiditing.DeleterUserId))
{
    // DeleterUserId will be set by ASP.NET Boilerplate automatically
}
```

Active/Passive Entities

Some entities need to be marked as Active or Passive. You may take an action upon the active/passive states of the entity. You can implement the **IPassivable** interface that has been created for this reason. It defines the **IsActive** property.

If your entity will be active on it's first creation, you can set IsActive to true in the constructor.

This is different than soft delete (IsDeleted). If an entity is soft deleted, it can not be retrieved from the database (ABP prevents it by default). For active/passive entities, it's completely up to you on how you control the retrieval of these entities.

Entity Change Events

ASP.NET Boilerplate automatically triggers certain events when an entity is inserted, updated or deleted. You can register to these events and perform any logic you need. See the Predefined Events section in the event bus documentation for more information.

IEntity Interfaces

The **Entity** class implements the **IEntity** interface (and **Entity<TPrimaryKey>** implements **IEntity<TPrimaryKey>**). If you do not want to derive from the Entity class, you can implement these interfaces directly. There are also corresponding interfaces for other entity classes. We don't recommend you do this unless you have a good reason not to derive from the Entity classes.

Multi-Lingual Entities

ASP.NET Boilerplate provides a simple way for defining and using Multi-Lingual entities. For more information see Multi-Lingual Entities.

IExtendableObject Interface

ASP.NET Boilerplate provides a simple interface, **IExtendableObject**, to easily associate **arbitrary name-value data** to an entity. Consider this simple entity:

Copy

```
public class Person : Entity, IExtendableObject
{
    public string Name { get; set; }

    public string ExtensionData { get; set; }

    public Person(string name)
    {
        Name = name;
    }
}
```

IExtendableObject defines the **ExtensionData** string property which is used to store **JSON formatted** name value objects. Example:

Copy

```
var person = new Person("John");

person.SetData("RandomValue", RandomHelper.GetRandom(1, 1000));
person.SetData("CustomData", new MyCustomObject { Value1 = 42, Value2 = "forty-two" });
```

We can use any type of object as a value to the **SetData** method. When we use the code above, **ExtensionData** will look like this:

Copy

```
{"CustomData":{"Value1":42,"Value2":"forty-two"},"RandomValue":178}
```

We can then use **GetData** to get the values:

Copy

```
var randomValue = person.GetData<int>("RandomValue");
var customData = person.GetData<MyCustomObject>("CustomData");
```

Of course, we can also use the **RemoveData** method to remove data:

Copy

```
person.RemoveData("RandomValue");
```

While this technique can be very useful in some cases (when you need to provide the ability to dynamically add extra data to an entity), you should normally use regular properties. Such dynamic usage is not type safe and explicit.