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

Default Repositories
Custom Repositories
Custom Repository Interface
Custom Repository Implementation
Base Repository Methods
Querying
About IQueryable<T>
Insert
Update
Delete
Others
About Async Methods
Managing Database Connections
Lifetime of a Repository
Repository Best Practices

Edit on GitHub

The repository pattern "*Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects*" (Martin Fowler).

Repositories, in practice, are used to perform database operations for domain objects (Entity and Value types). Generally, a separate repository is used for each Entity (or Aggregate Root).

Default Repositories

In ASP.NET Boilerplate, repository classes implement the **IRepository<TEntity, TPrimaryKey>** interface. ABP can automatically create default repositories for each entity type. You can directly inject **IRepository<TEntity>** (or IRepository<TEntity, TPrimaryKey>). An example application service uses a repository to insert an entity into a database:

Copy

```
public class PersonAppService : IPersonAppService
{
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public void CreatePerson(CreatePersonInput input)
    {
        person = new Person { Name = input.Name, EmailAddress = input.EmailAddress };
        _personRepository.Insert(person);
    }
}
```

The PersonAppService contructor-injects **IRepository<Person>** and uses the **Insert** method.

Authorization
Feature Management
Audit Logging
Entity History
Distributed Service Layer
ASP.NET Web API
Web API Controllers
Dynamic Web API Layer
OData Integration

Custom Repositories

You only create a repository class for an entity when you need to create custom repository methods for that entity.

Custom Repository Interface

A repository definition for a Person entity is shown below:

Copy

```
public interface IPersonRepository : IRepository<Person>
{

}
```

IPersonRepository extends **IRepository<TEntity>**. It's used to define entities which have a primary key type of int (Int32). If your entity's primary key is not an int, you can extend the **IRepository<TEntity, TPrimaryKey>** interface as shown below:

Copy

```
public interface IPersonRepository : IRepository<Person, long>
{

}
```

Custom Repository Implementation

ASP.NET Boilerplate is designed to be independent from a particular ORM (Object/Relational Mapping) framework or another technique to access a database. Repositories are implemented in **NHibernate** and **EntityFramework**, out-of-the-box. See the following documents to implement repositories in ASP.NET Boilerplate on these frameworks:

- NHibernate integration
- EntityFramework integration

Base Repository Methods

Every repository has some common methods coming from the IRepository<TEntity> interface. We will investigate most of them here.

Querying

Getting single entity

Copy

```
TEntity Get(TPrimaryKey id);
Task<TEntity> GetAsync(TPrimaryKey id);
TEntity Single(Expression<Func<TEntity, bool>> predicate);
Task<TEntity> SingleAsync(Expression<Func<TEntity, bool>> predicate);
TEntity FirstOrDefault(TPrimaryKey id);
Task<TEntity> FirstOrDefaultAsync(TPrimaryKey id);
TEntity FirstOrDefault(Expression<Func<TEntity, bool>> predicate);
Task<TEntity> FirstOrDefaultAsync(Expression<Func<TEntity, bool>> predicate);
TEntity Load(TPrimaryKey id);
```

The **Get** method is used to get an Entity with a given primary key (Id). It throws an exception if there is no entity in the database with the given Id. The **Single** method is similar to Get but takes an expression rather than an Id. This way, you can write a lambda expression to get an Entity. Example usages:

Copy

```
var person = _personRepository.Get(42);
var person = _personRepository.Single(p => p.Name == "John");
```

Note that the **Single** method throws an exception if there is no entity with the given conditions or where there is more than one entity.

Instead of throwing an exception, **FirstOrDefault** is similar but returns **null** if there is no entity with a given Id or expression. It returns the first found entity if there are more than one entity for the given conditions.

Load does not retrieve the entity from the database but creates a proxy object for lazy-loading. If you only use the Id property, the Entity is not actually retrieved. It's retrieved from the database only if you access other properties of the entity. This can be used instead of Get, for performance reasons. It's implemented in **NHibernate**. If the ORM provider does not implements it, the Load method works identically to the Get method.

Getting a list of entities

Copy

```
List<TEntity> GetAllList();
Task<List<TEntity>> GetAllListAsync();
List<TEntity> GetAllList(Expression<Func<TEntity, bool>> predicate);
Task<List<TEntity>> GetAllListAsync(Expression<Func<TEntity, bool>> predicate);
IQueryable<TEntity> GetAll();
IQueryable<TEntity> GetAllIncluding(params Expression<Func<TEntity, object>>[]
propertySelectors)
```

The **GetAllList** method is used to retrieve all entities from the database. An overload of it can be used to filter entities. Examples:

Copy

```
var allPeople = _personRepository.GetAllList();
var somePeople = _personRepository.GetAllList(person => person.IsActive && person.Age >
42);
```

The **GetAll** method returns an IQueryable<T>. This way, you can add Linq methods after it. Examples:

Copy

```
//Example 1
var query = from person in _personRepository.GetAll()
            where person.IsActive
            orderby person.Name
            select person;
var people = query.ToList();

//Example 2:
List<Person> personList2 = _personRepository.GetAll().Where(p =>
p.Name.Contains("H")).OrderBy(p => p.Name).Skip(40).Take(20).ToList();
```

When using GetAll, almost all queries can be written in Linq. It can even be used in a join expression!

The **GetAllIncluding** allows to specify related data to be included in query results. In the following example, people will be retrieved with their Addresses property populated.

Copy

```
var allPeople = await _personRepository.GetAllIncluding(p => p.Addresses).ToListAsync();
```

About IQueryable<T>

When you call GetAll() out of a repository method, there must be an open database connection. This is because of the deferred execution of IQueryable<T>. It does not perform a database query unless you call the ToList() method or use the IQueryable<T> in a foreach loop (or somehow access the queried items). So when you call the ToList() method, the database connection must be alive. For a web application, you don't need to worry about that in most cases since the MVC controller methods are units of work by default and the database connection is available for the entire request. See the **UnitOfWork** documentation to understand it better.

Custom return value

There is also an additional method to provide the power of the IQueryable that can be usable out of a unit of work.

```
T Query<T>(Func<IQueryable<TEntity>, T> queryMethod);
```

The Query method accepts a lambda (or method) that receives IQueryable<T> and returns any type of object. Example:

```
var people = _personRepository.Query(q => q.Where(p => p.Name.Contains("H")).OrderBy(p => p.Name).ToList());
```

Since the given lamda (or method) is executed inside the repository method, it's executed when the database connection is available. You can return a list of entities, a single entity, or a projection or something else that executes the query.

Insert

The IRepository interface defines methods to insert an entity to database:

```
TEntity Insert(TEntity entity);
Task<TEntity> InsertAsync(TEntity entity);
TPrimaryKey InsertAndGetId(TEntity entity);
Task<TPrimaryKey> InsertAndGetIdAsync(TEntity entity);
TEntity InsertOrUpdate(TEntity entity);
Task<TEntity> InsertOrUpdateAsync(TEntity entity);
TPrimaryKey InsertOrUpdateAndGetId(TEntity entity);
Task<TPrimaryKey> InsertOrUpdateAndGetIdAsync(TEntity entity);
```

The **Insert** method simply inserts new a entity in to a database and returns the same inserted entity. The **InsertAndGetId** method returns the Id of a newly inserted entity. This is useful if the Id is auto-increment and you need the Id of the newly inserted entity. The **InsertOrUpdate** method inserts or updates a given entity by checking its Id's value. Lastly, the **InsertOrUpdateAndGetId** method returns the Id of the entity after inserting or updating it.

Update

The IRepository interface defines methods to update an existing entity in the database. It takes the entity to be updated and returns the same entity object.

```
TEntity Update(TEntity entity);
Task<TEntity> UpdateAsync(TEntity entity);
```

Most of the time you don't need to explicitly call the Update methods since the unit of work system automatically saves all changes when the unit of work completes. See the unit of work documentation for more info.

Delete

The IRepository interface defines methods to delete an existing entity from the database

```
void Delete(TEntity entity);
Task DeleteAsync(TEntity entity);
void Delete(TPrimaryKey id);
Task DeleteAsync(TPrimaryKey id);
void Delete(Expression<Func<TEntity, bool>> predicate);
Task DeleteAsync(Expression<Func<TEntity, bool>> predicate);
```

The first method accepts an existing entity, the second one accepts an Id of the entity to delete. The last one accepts a condition to delete all entities that fit a given condition. Note that all entities matching a given predicate may be retrieved from the database and then deleted (based on repository implementation). So use it carefully! It may cause performance problems if there are too many entities with a given condition.

Others

The IRepository also provides methods to get the count of entities in a table.

Copy

```
int Count();
Task<int> CountAsync();
int Count(Expression<Func<TEntity, bool>> predicate);
Task<int> CountAsync(Expression<Func<TEntity, bool>> predicate);
long LongCount();
Task<long> LongCountAsync();
long LongCount(Expression<Func<TEntity, bool>> predicate);
Task<long> LongCountAsync(Expression<Func<TEntity, bool>> predicate);
```

About Async Methods

ASP.NET Boilerplate supports an async programming model. The repository methods have async versions.

Here's a sample application service method that uses the async model:

Copy

```
public class PersonAppService : AbpWpfDemoAppServiceBase, IPersonAppService
{
    private readonly IRepository<Person> _personRepository;

    public PersonAppService(IRepository<Person> personRepository)
    {
        _personRepository = personRepository;
    }

    public async Task<GetPeopleOutput> GetAllPeople()
    {
        var people = await _personRepository.GetAllListAsync();

        return new GetPeopleOutput
        {
            People = Mapper.Map<List<PersonDto>>(people)
        };
    }
}
```

The GetAllPeople method is async and uses GetAllListAsync with the await keyword.

Async may not be supported by all ORM frameworks. It's supported by EntityFramework. If it's not supported, the Async repository methods work synchronously. For example, InsertAsync works the same as Insert in EntityFramework since EF does not write new entities to the database until the unit of work completes (a.k.a. DbContext.SaveChanges).

Managing Database Connections

A database connection is not opened or closed in a repository method. Connection management is made automatically by ASP.NET Boilerplate.

A database connection is **opened** and a **transaction** automatically begins while entering a repository method. When the method ends and returns, all changes are **saved**, the transaction is **committed** and the database connection is **closed** by ASP.NET Boilerplate. If your repository method throws any type of Exception, the transaction is automatically **rolled back** and the database connection is closed. This is true for all public methods of classes that implement the IRepository interface.

If a repository method calls another repository method (even a method of a different repository) they share the same connection and transaction. The connection is managed (opened/closed) by the first method that enters a repository. For more information on database connection management, see the UnitOfWork documentation.

Lifetime of a Repository

All repository instances are **Transient**. This means they are instantiated per-usage. See the Dependency Injection documentation for more information.

Repository Best Practices

- For an entity of T, use IRepository<T> wherever it's possible. Don't create custom repositories unless it's really needed. The pre-defined repository methods will be enough for most cases.
- If you are creating a custom repository (by extending IRepository<TEntity>);
  - Repository classes should be stateless. That means you must not define repository-level state objects and a repository method call should not effect another call.
  - Custom repository methods should not contain business logic or application logic. It should just perform data-related or orm-specific tasks.
- While repositories can use dependency injection, define fewer or no dependencies to other services.