Repository Pattern

# Repository

## What is a Repository?

“A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection.” (Martin Fowler)

First of all, let’s state the obvious:

* A Repository class is called like so because it is a place where the application stores its domain entities. And I want to emphasize this: domain entities; not database entities, not DTOs, not View Models or any other type of objects, but domain entities.

Each Repository class must contain a single type of domain entity. In other words, there must be created a Repository class for each type of domain entity.

## Benefits

* **Use the storage like an in-memory collection**
  + Allows the rest of the application to work with the data as it would be an in-memory collection.
  + The repository should provide Get, Add, Remove, but no Update method.
  + No Save method, also.
    - The Save method should be present in the Unit of Work class.
* **Hides the details of accessing the data storage.**
  + The data may be stored in a database, a file on disk (xml, json, binary, etc...) or even a web service. The repository class encapsulates the details that come with each storage type.
  + The framework used to access the data (like Entity Framework, NHibernate, Dapper, ADO.NET, etc.) is also a detail.
* **Eliminates duplicate query logic.**
  + This is the place where we can write queries that will be used, later, in multiple places in the rest of the application.
  + Indeed, there is also other ways of avoiding code duplication without using repository classes. For example, to create extended methods for DbSet<T> where to write those queries.

# Unit of Work

## What is a Unit of Work?

“The Unit of Work maintains a list of objects affected by a business transaction and coordinates the writing out of changes.” (Martin Fowler)

## Benefits

* **Ensure data consistency**
  + When modifications are performed on multiple repositories, the Unit of Work is ensuring that all the changes, from all the repositories, are ether all successfully saved, or none are saved into the data storage.
* **Allows “rollback” of the unsaved modifications**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **Transaction** | **Data Storage**  **Location** | **Frameworks**  **(Suggested)** | **Repository Class** | | **Unit of Work Class** |
| **Update()** | **Save()** | **Save()** |
| 1 | - | in-memory | None | No | No | - |
| 2 | - | out-of- memory | ADO.NET | Yes | No | - |
| 3 | Repository  Transaction | out-of- memory | Entity Framework | No | Yes | - |
| 4 | Cross-Repository  Transaction | out-of- memory | Entity Framework | No | No | Yes |

# Demo 1 – Repository with in-memory data

## Characteristics

* The data is stored in static collections, in memory.
* No Update or Save methods are needed.

## Can it be used like an in-memory collection?

* **Yes**
* The application can retrieve entities from the repository and update them as needed.

|  |  |  |
| --- | --- | --- |
| Operation | Method | Description |
| Get | IEnumerable<Product> GetAll();  Product Get(int id); | Situations:   * Retrieve all entities. * Retrieve entity by id. * Other custom Get methods. |
| Add | void Add(Product product); | Steps:   1. Create a new entity. 2. Add it to repository. |
| Remove | void Delete(int id);  void Delete(Product product); | Situations:   * If we have the entity’s id:   + Ask to remove the entity with that id. * If we have the entity:   + Ask to remove the entity. |
| Update |  | Steps:   1. Retrieve an entity. 2. Update its fields. |

Note: Use an application with Clean Architecture (Domain – Application – Data Access)

# Demo 2 – Out-of-memory data storage

## Characteristics

If we place the data into a different location than memory, for example on the disk, we suddenly need an Update method in order to persist the changes made to the entities.

## Can it be used like an in-memory collection?

* **No**
* When updating the entities retrieved from the Repository, the modifications are not automatically saved in the persistent storage.

|  |  |  |
| --- | --- | --- |
| Operation | Method | Description |
| Get | IEnumerable<Product> GetAll();  Product Get(int id); | Situations:   * Retrieve all entities. * Retrieve entity by id. * Retrieve entities based on a filter. |
| Add | void Add(Product product); | Steps:   1. Create a new entity. 2. Add it to repository. |
| Remove | void Delete(int id);  void Delete(Product product); | Situations:   * If we have the entity’s id:   + Ask to remove the entity with that id. * If we have the entity:   + Ask to remove the entity. |
| Update | void Update(Product product); | Steps:   1. Create an entity with the correct id and other fields. 2. Send it to repository. |

Use ADO.NET

# Demo 3 – Repository-level transaction

## Characteristics

Instead of having an Update method, the Repository implements a mechanism to track any changes to the entities and persists them only when the Save method is called.

We can implement our own mechanism of tracking the changes or we can use an already existing one, like the one from Entity Framework.

## Can it be used like an in-memory collection?

* **Yes**
* The only missing feature is that it does not provide cross-repository transactions.

|  |  |  |
| --- | --- | --- |
| Operation | Method | Description |
| Get | IEnumerable<Product> GetAll();  Product Get(int id); | Situations:   * Retrieve all entities. * Retrieve entity by id. * Retrieve entities based on a filter. |
| Add | void Add(Product product); | Steps:   1. Create a new entity. 2. Add it to repository. |
| Remove | void Delete(int id);  void Delete(Product product); | Situations:   * If we have the entity’s id:   + Ask to remove the entity with that id. * If we have the entity:   + Ask to remove the entity. |
| Update | Repository:  void Save(); | Steps:   1. Retrieve one or more entities from the same repository. 2. Update their fields. 3. Call Save method. |

# Demo 4 - Cross-repository transaction

## Characteristics

A Unit of Work class is needed to keep track of the changes to every entity from every repository. Again, we can implement our own mechanism, or we can use the already existing one from Entity Framework (the DbContext class).

## Can it be used like an in-memory collection?

* **Yes**
* We can update the entities retrieved from the repository and we have cross-repository transactions.

|  |  |  |
| --- | --- | --- |
| Operation | Method | Description |
| Get | IEnumerable<Product> GetAll();  Product Get(int id); | Situations:   * Retrieve all entities. * Retrieve entity by id. * Retrieve entities based on a filter. |
| Add | void Add(Product product); | Steps:   1. Create a new entity. 2. Add it to repository. |
| Remove | void Delete(int id);  void Delete(Product product); | Situations:   * If we have the entity’s id:   + Ask to remove the entity with that id. * If we have the entity:   + Ask to remove the entity. |
| Update | UnitOfWork:  void Save(); | Steps:   1. Retrieve entities from one or more repositories. 2. Update their fields. 3. Call Save method on the UnitOfWork. |

# Entity Framework vs Custom Repository and Unit of Work

## Entity Framework

1. DbSet<T> is a Repository
   * (+) Hides the details of accessing the data
   * (+) Emulates an in-memory collection
   * (-) We have no place where to write custom queries that can be later used in multiple places in the rest of the application.
2. DbContext is a Unit of Work
   * (+) Ensures data consistency
     + It provides the SaveChanges method.

## Important Question

Do we really need to create our custom Repository and UnitOfWork classes if we already use the Entity Framework?

This is a decision that must be taken by each team on each project, but please, take it consciously. Make a meeting, discuss the subject, talk about advantages and disadvantages, and then take the decision.

Here are some advantages and disadvantages that I can think of for creating custom Repository and Unit of Work classes.

### Advantages

* Encapsulates (hides) the usage of Entity Framework.
  + Entity Framework is a detail of the Data Access Layer. None of the other modules of the application should be aware of its existence. They should not be able to tell if Entity Framework, NHibernate, Dapper, ADO.NET or any other data access technology is used.
  + By doing so, it allows us to easily test the other modules, make updates to the Entity Framework or replace it with another mechanism.
* Helps to avoid code duplication.
  + The Repository classes offers a place where we can write queries that are needed in multiple places in the application.
  + There are, indeed, other ways to avoid this code duplication. For example, by creating extended methods for DbSet<T> where the queries can be placed.

### Disadvantages

* Creating an additional abstraction layer, adds complexity to the application.

# Demo 1 - No Entity Framework

In-memory collection of data.

Repository classes.

No Unit of Work.

# Demo 2 – With Entity Framework (no Repository, no Unit of Work)

Directly use DbContext and DbSet<T> in the application.

Show situation where queries are duplicated in the code.

### Solution 1

* Create a Repository class where to put the queries.
* DbContext must be received on constructor.

### Solution 2

* Create an extension method on DBSet<T> for each query.

# Demo 3 – With Entity Framework (with Repository and Unit of Work)

Two repositories that must be updated in a consistent way

### Solution 1

* Use DbContext.SaveChanges() method. – Done

### Solution 2

* Use Unit of Work