Repository Pattern

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **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. |

# 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