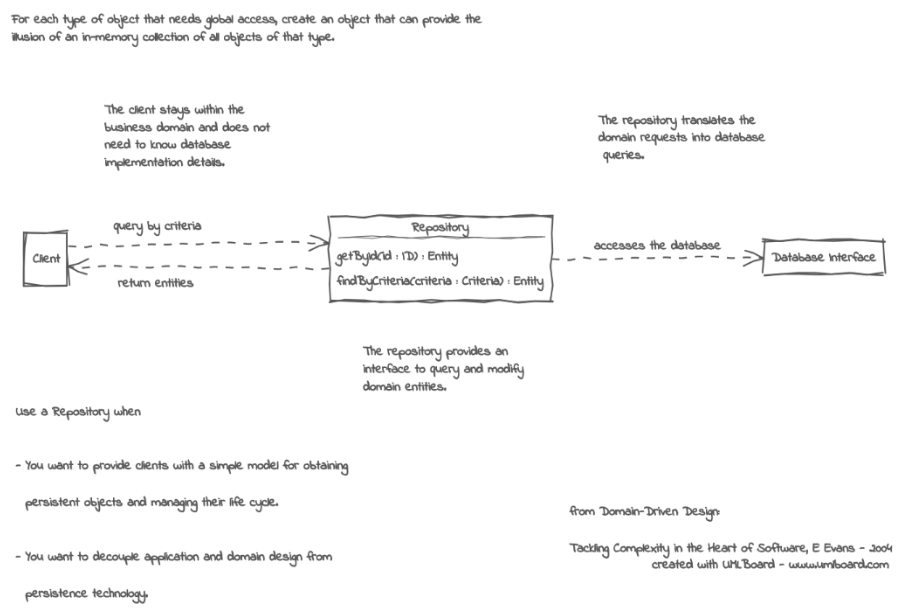
Spring Data Repositories Lab

The Repository is a central design pattern of the Domain-Driven-Design (DDD) philosophy. The idea behind DDD is to make the business model and its domain concepts the central aspect of your software design -- instead of earlier strategies where the main focus was the technology and the implementation. In this context, the *Repository* aims to decouple the domain logic from the underlying database layer, thus making your architecture more flexible and easier to maintain.


For each type of object that needs global access, create an object that can provide the illusion of an in-memory collection of all objects of that type.
The client stays within the business domain and does not need to know database implementation details.
The repository translates the domain requests into database queries.
query by criteria
Client
return entities
Repository
getById(id : ID) : Entity
findByCriteria(criteria : Criteria) : Entity
accesses the database
Database Interface
The repository provides an interface to query and modify domain entities.
Use a Repository when
- You want to provide clients with a simple model for obtaining persistent objects and managing their life cycle.
- You want to decouple application and domain design from persistence technology.
from Domain-Driven Design:
Tackling Complexity in the Heart of Software, E Evans - 2004
created with UMLBoard - www.umlboard.com

A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

The Repository pattern makes it easier to test your application logic

The Repository pattern allows you to easily test your application with unit tests. Remember that unit tests only test your code, not infrastructure, so the repository abstractions make it easier to achieve that goal.

it's recommended that you define and place the repository interfaces in the domain model layer so the application layer, such as your Web API microservice, doesn't depend directly on the infrastructure layer where you've implemented the actual repository classes. By doing this and using Dependency Injection in the controllers of your Web API, you can implement mock repositories that return fake data instead of data from the database. This decoupled approach allows you to create and run unit tests that focus the logic of your application without requiring connectivity to the database.

In terms of separation of concerns for unit tests, your logic operates on domain entities in memory. It assumes the repository class has delivered those. Once your logic modifies the domain entities, it assumes the repository class will store them correctly. The important point here is to create unit tests against your domain model and its domain logic. Aggregate roots are the main consistency boundaries in DDD.

The difference between the Repository pattern and the legacy Data Access class (DAL class) pattern

A typical DAL object directly performs data access and persistence operations against storage, often at the level of a single table and row. Simple CRUD operations implemented with a set of DAL classes frequently do not support transactions (though this is not always the case). Most DAL class approaches make minimal use of abstractions, resulting in tight coupling between application or Business Logic Layer (BLL) classes that call the DAL objects.

When using repository, the implementation details of persistence are encapsulated away from the domain model. The use of an abstraction provides ease of extending behavior through patterns like Decorators or Proxies. For instance, cross-cutting concerns like caching, logging, and error handling can all be applied using these patterns rather than hard-coded in the data access code itself. It's also trivial to support multiple repository adapters which may be used in different environments, from local development to shared staging environments to production.

SPRING DATA REPOSITORIES

Simply put, every repository in Spring Data extends the generic *Repository* interface, but beyond that, they each have different functionality. *Starting with the JpaRepository* – which extends *PagingAndSortingRepository* and, in turn, the *CrudRepository*.

Each of these defines its functionality:

- *CrudRepository* provides CRUD functions
- *PagingAndSortingRepository* provides methods to do pagination and sorting of records
- *JpaRepository* provides JPA-related methods such as flushing the persistence context and deleting records in a batch

And so, because of this inheritance relationship, the ***JpaRepository* contains the full API of *CrudRepository* and *PagingAndSortingRepository*.**

When we don't need the full functionality provided by *JpaRepository* and *PagingAndSortingRepository*, we can use the *CrudRepository*.

PagingAndSortingRepository

```
public interface PagingAndSortingRepository<T, ID extends Serializable>
 extends CrudRepository<T, ID> {


  Iterable<T> findAll(Sort sort);


  Page<T> findAll(Pageable pageable);
}
```

This interface provides a method *findAll(Pageable pageable)*, which is the key to implementing *Pagination*.

When using *Pageable*, we create a *Pageable* object with certain properties, and we've to specify at least the following:

1. Page size
2. Current page number

3. Sorting

So, let’s assume that we want to show the first page of a result set sorted by *lastName,* ascending, having no more than five records each. This is how we can achieve this using a *PageRequest* and a *Sort* definition:

```
Sort sort = new Sort(new Sort.Order(Direction.ASC, "lastName"));

Pageable pageable = new PageRequest(0, 5, sort);
```

Passing the pageable object to the Spring data query will return the results in question (the first parameter of *PageRequest* is zero-based).

5. *JpaRepository*

Finally, we’ll have a look at the *JpaRepository* interface:

```
public interface JpaRepository<T, ID extends Serializable> extends

  PagingAndSortingRepository<T, ID> {


    List<T> findAll();


    List<T> findAll(Sort sort);


    List<T> save(Iterable<? extends T> entities);


    void flush();


    T saveAndFlush(T entity);


    void deleteInBatch(Iterable<T> entities);

}
```

Again, let’s look at each of these methods in brief:

- *findAll()* – get a *List* of all available entities in the database

- *findAll(…)* – get a *List* of all available entities and sort them using the provided condition
- *save(…)* – save an *Iterable* of entities. Here, we can pass multiple objects to save them in a batch
- *flush()* – flush all pending tasks to the database
- *saveAndFlush(…)* – save the entity and flush changes immediately
- deleteInBatch(…) – delete an *Iterable* of entities. Here, we can pass multiple objects to delete them in a batch

Clearly, the above interface extends *PagingAndSortingRepository,* which means it also has all methods present in the *CrudRepository*.

References

- Catalog of Patterns of Enterprise Application Architecture: Repository
- Design the infrastructure persistence layer
- The Downsides Of Using The Repository Pattern
- The generic repository is just a lazy anti-pattern
- Implementing the Repository and Unit of Work Patterns in an ASP.NET MVC Application