"Repository commonly refers to a storage location, often for safety or preservation." - Wikipedia

This is how Wikipedia describes a repository. It just so happens that unlike some other jargon that we deal with, this fits perfectly. A repository represents the concept of a storage collection for a specific type of entity.

Repository as a Collection

Probably the most important distinction about repositories is that they represent collections of entities. They do not represent database storage or caching or any number of technical concerns. Repositories represent collections. How you hold those collections is simply an implementation detail.

I want to make this point clear. A repository is a collection, a collection that holds entities and which can filter and return entities back based on the needs of your application. How it actually holds those entities is an IMPLEMENTATION DETAIL.

In the world of PHP we're used to the Request / Response cycle followed by the death of the PHP process. Anything that isn't persisted externally is gone for good, at this point. Now, not all platforms work this way.

I find that a good thought-experiment to help understand repositories is to imagine that your application is always running and that objects always stay in memory. We're not worried about critical failures for this experiment. Imagine that you have a singleton repository for Member entities, MemberRepository.

Then, you create a new Member and add it to the repository. Later, you ask the repository for all members and you receive back a collection that contains the Member you added. Perhaps you want to pull back a specific member by ID, you can do that too. It's easy to imagine that inside the repository these Member objects are stored in an array or better yet a collection object.

Simply put, a repository is a special kind of authoritative collection that you'll use over and over to hold and filter back entities.

Interacting with a Repository

Imagine that we create a Member entity. We become satisfied with the Member object, then the request ends and the Member object disappears. The member then tries to login to our app, and they can't. Obviously, we need to make the Member available to other parts of our application.

```php
<?php

$member = Member::register($email, $password);
$memberRepository->save($member);
```

Now, we can access the member later. Perhaps something like...

```php
<?php

$member = $memberRepository->findByEmail($email);
// or
$members = $memberRepository->getAll();
```

Now, we can store Member objects in one part of our application and then retrieve them from a separate part.

Should Repositories Create Entities?

You may have come across examples like this:

```php
<?php

$member = $memberRepository->create($email, $password);
```

I've seen people make arguments for this. But, I am quite uninterested in this approach.

For one, repositories are collections. I'm not sure why the collection should be a collection*and* a factory. I've heard arguments like.. if it's handling persistence, why not handle creation?

In my mind, this is an anti-pattern. Why not allow the Member to have its own understanding of its creation or why not have a factory that is specifically designed to put together more complex objects?

If we treat our repositories as simple collections then we are giving them a single responsibility. I don't want collection classes that are also factories.

What is the Benefit of Repositories?

The primary benefit of repositories is to abstract the storage mechanism for the authoritative collection of entities.

By providing a MemberRepository interface we allow any number of concretions to implement it.

```php
<?php

interface MemberRepository {
    public function save(Member $member);
    public function getAll();
    public function findById(MemberId $memberId);
}
```

```php
<?php

class ArrayMemberRepository implements MemberRepository {
    private $members = [];

    public function save(Member $member) {
        $this->members[(string)$member->getId()] = $member;
    }

    public function getAll() {
        return $this->members;
    }

    public function findById(MemberId $memberId) {
        if (isset($this->members[(string)$memberId])) {
            return $this->members[(string)$memberId];
        }
    }
}
```

```php
<?php

class RedisMemberRepository implements MemberRepository {
    public function save(Member $member) {
        // ...
    }

    // you get the point
}
```

In this way, most of our application knows only of the abstract concept of a MemberRepository and our usage of it can be decoupled from the actual implementation. This is quite liberating.

Do Repositories Belong to Domain or Application Service Layers?

Now, here's an interesting question. First, let's define the application service layer as the layer in a multi-layered architecture that is responsible for application-specific implementation details such as database persistence, internet protocol knowledge (sending email, API interactions), etc.

Let's define the domain layer as the layer of a multi-layered architecture that is responsible for holding business rules and business logic.

Working with these definitions, where do our repositories fit?

Let's take a look at our example source from earlier.

```php
<?php

class ArrayMemberRepository implements MemberRepository {
    private $members = [];

    public function save(Member $member) {
        $this->members[(string) $member->getId()] = $member;
    }

    public function getAll() {
        return $this->members;
    }

    public function findById(MemberId $memberId) {
        if (isset($this->members[(string)$memberId])) {
            return $this->members[(string)$memberId];
        }
    }
}
```

In this example, I am seeing a lot of implementation detail. This implementation detail clearly belongs in the application-service layer.

For shits and grins, let's remove all of the application detail from that class..

```
<?php

class ArrayMemberRepository implements MemberRepository {
    public function save(Member $member) {
    }

    public function getAll() {
    }

    public function findById(MemberId $memberId) {
    }
}
```

Hmmm... this is starting to look familiar.. What are we left with?

Maybe it reminds you of this?

```
<?php

interface MemberRepository {
    public function save(Member $member);
    public function getAll();
    public function findById(MemberId $memberId);
}
```

This is what it means to place an interface at layer boundaries. This interface actually can contain domain-specific concepts, the implementation of the interface should not.

Repository interfaces belong to the domain-layer. The implementation of repositories belong to the application-service layer. This means that we're free to type-hint for our repositories in our domain-layer without ever having to depend on the service layer.

Freedom to Swap Out Data Stores

Whenever you hear someone talking about object-oriented design concepts, you probably hear something like, "and you have the freedom to swap one data store out for another at a later date."

I've come to realize that while this is not entirely untrue.. it IS a VERY poor argument. The biggest problem with this explanation is that it begs the question, "are you really going to want to do that?" I do NOT want that question to be what determines whether or not someone wants to use the repository pattern.

Any sufficiently designed object-oriented application automatically comes with this type of advantage. A very central concept to object-orientation is encapsulation. You can expose an API and hide implementation.

The truth is, you probably won't switch from one ORM to another. But, if you do at least you have a reasonable way to begin. However, swapping out implementations of a repository is fantastic for testability.

Testability with the Repository Pattern

Well, guess what. It's pretty sweet. Let's say that you have an object that handles something like member registration..

```php
<?php

class RegisterMemberHandler {
    private $members;

    public function __construct(MemberRepository $members) {
        $this->members = $members;
    }
    public function handle(RegisterMember $command) {
        $member = Member::register($command->email, $command->password);
        $this->members->save($member);
    }
}
```

During regular operations, I can inject an instance of a DoctrineMemberRepository. However, during testing I can easily inject an instance of ArrayMemberRepository. They both implement the interface.

A simplified test example might look something like this...

```php
<?php

$repo = new ArrayMemberRepository;
$handler = new RegisterMemberHandler($repo);

$request = $this->createRequest(['email' => 'bob@bob.com', 'password' =>
'angelofdestruction']);

$handler->handle(RegisterMember::usingForm($request));

AssertCount(1, $repo->findByEmail('bob@bob.com'));
```

Is this example, we're testing the handler. We don't need to test that the repository stored the data in the database (or wherever). We need to test the specific behavior of this object, which is to ask the Member class for a new member based on arguments from the command, then pass it into a repository.

Collection-Oriented vs Persistence-Oriented

In the book Implementing Domain-Driven Design Vaughn Vernon makes a distinction between collection-based and persistence-based repositories. In short, the idea of a collection-oriented repository is that the repository is treated as an in-memory array store. But, a persistence-oriented repository comes with an assumption that there's going to be some deeper storage going on. This plays out mostly in naming.

```php
<?php

// collection-oriented
$memberRepository->add($member);

// vs persistence-oriented
$memberRepository->save($member);
```

I don't currently have much of an opinion on which is used. However, I reserve the right to change. In the end, I focus on them as collection objects with the same responsibility that any other collection object might have.

Summary

I believe that..

- ..it's important to give repositories the singular task of functioning as collection objects.
- ..we shouldn't use repositories to create new object instances.
- ..we should avoid selling decoupling as a way to transition from one technology to another as it has so many other benefits that are harder to dismiss.

In the future, I intend to write a few more articles relating to repositories such as caching repository results using the decorator pattern, querying using the criteria pattern, the repository's role in guarding aggregate invariants, and handling batch operations on large numbers of objects.

If you have questions or if your views differs from mine, please comment below.

As always, I intend to change this article to sync up with my opinions.