Creating a User Registration Domain Service

PHILIP BROWN
20 OCT 2014 • 8 MIN READ

Nearly all consumer facing web applications will require registration functionality in one form or another. The registration process is often the first contact that the user will have with the internal flow of an application. It's therefore very important that you make a good impression or the user might never give your application a second chance.

There are many different ways for implementing a registration process. In a lot of web applications, the registration process will be just a simple form, whilst in other applications it will be an intricate flow consisting of many different screens.

In today's tutorial we are going to look at building out a registration process in the form of a Domain Service. We will recap the characteristics of Domain Services and we will leverage many of the objects and services from the last couple of weeks.

When we looked at Domain Services a couple of weeks ago, an important takeaway was, you should try to avoid using Domain Services unless they are truly necessary.

Domain Services can often steal responsibility away from the Domain Objects. This leaves you in a situation where your Domain Objects are basically just bags of getters and setters, and all of your Domain logic is wrapped up in services.

Domain Services should only be used when the functionality does not naturally fit on an existing Domain object, the functionality is stateless and it will require a co-ordinated effort of a number of Domain Objects and Services.

A couple of weeks ago we looked at using the ubiquitous language within our code. This lead us to using a static `register()` method to create a new `User`.

So we already have a method on an existing Domain object. Shouldn't this functionality be encapsulated on the `User` object?

I believe, in the context of Cribbb, that the registration functionality should be a Domain Service for a couple of reasons.

Firstly, an important criteria of the business rules of the application state that all username and email addresses should be unique. In order to check for uniqueness we need to query the database. The `User` object should not be concerned with checking the database.

`er` object either.

So we are faced with a situation where the responsibility of registration new users requires the ability to check the database as well as hashing a password.

This functionality should clearly not be jammed into the `User` object as the `User` object should not have the ability to query the database or hash passwords.

With that being said, I think for many applications the functionality of registering a new user should be encapsulated on the `User` object. If your registration process is not very complicated, abstracting that logic away from the `User` object would be a mistake.

Why isn't this an Application Service?

If you are familiar with the architecture of Domain Driven Design applications, you might be thinking that the registration functionality should be modelled as an Application Service.

We haven't really covered Application Services so far in this series, but we will in the next couple of weeks.

Application Services control the flow of data as it enters and leaves your application. You can think of Application Services as providing your application with a public API to the outside world.

For the most part I would tend to agree that the registration process for a web application could

The Application Service is responsible for accepting the request from the outside world, processing it, and then returning a response. I think allowing the Application Service to co-ordinate the various Domain Services and Domain Objects required to register a new user would be a fine solution.

However I'm still going to be modelling it as a Domain Service in Cribbb.

Generally speaking, I want to capture as much of the functionality as I can in Domain Services. Pushing important logic like the registration process out to the Application Services feels to me like I'm losing a bit of the control that I would have if I kept things close to the heart of the application.

If you feel like the architecture of your application makes sense to have the registration process as an Application Service, you should definitely do that. There is no hard and fast rule about what should be a Domain Service verses what should be an Application Service across all types of web application.

Create the Register User Service class

So the first thing to do will be to create the `RegisterUserService`. This class will live under the `Domain/Services/Identity` namespace. I've decided to split the Domain Services and Domain Model in to two separate namespaces to keep things manageable:

```
{
/**
* Register a new User
*
* @param string $email
* @param string $username
* @param string $password
* @return void
*/
public function register($email, $username, $password){}
}
```

The `RegisterUserService` class should have a single `register()` method that accepts an `$email`, `$username` and `$password`.

At this point I can also create the associated test class:

```
<?php namespace Cribbb\Tests\Domain\Services;

class RegisterUserServiceTest extends \PHPUnit_Framework_TestCase {}
```

Create the Value Objects

rules we have in place for this data.

```
$email = new Email($email);
$username = new Username($username);
$password = new Password($password);
```

If any of the raw input strings do not match the Value Object's internal business logic, an exception will be thrown.

Check for uniqueness

Next we need to check to ensure that the email and username have not already been registered. Fortunately we've already solved this problem when we looked at implementing The Specification Pattern.

The Specification Pattern will return `false` if the specification rule is not satisfied. However, in this situation we want to immediately jump out of the registration process if one of the values is not unique. To achieve this we can wrap each specification check in a method that will throw an exception if the specification is not satisfied:

```
/**
```

```
* @throws ValueIsNotUniqueException
* @return void
*/
private function checkEmailIsUnique(Email $email)
{
$specification = new EmailIsUnique($this->userRepository);

if(! $specification->isSatisfiedBy($email)) {
throw new ValueIsNotUniqueException("$email is already registered");
}
}

/**
* Check that a Username is unique
*
* @param Username $username
* @throws ValueIsNotUniqueException
* @return void
*/
private function checkUsernameIsUnique(Username $username)
{
$specification = new UsernameIsUnique($this->userRepository);

if(! $specification->isSatisfiedBy($username)) {
throw new ValueIsNotUniqueException("$username has already been taken");
}
}
```

Each of these methods requires access to the `$this->userRepository`. We can make this available by

```
/**
* @var UserRepository
*/
private $userRepository;

/**
* Create a new RegisterUserService
*
* @param UserRepository $userRepository
* @return void
*/
public function __construct(UserRepository $userRepository)
{
$this->userRepository = $userRepository;
}
```

We can now call the two `private` methods from the `register()` method. If an exception is thrown, the registration process will immediately halt:

```
$this->checkEmailIsUnique($email);
$this->checkUsernameIsUnique($username);
```

To make sure this is working correctly we can write two tests that expect that the exception should

First write a `setUp()` method that will create a new instance of the `RegisterUserRepository` and inject a mock of the `UserRepository`:

```
/** @var UserRepository */
private $repository;

/** @var RegisterUserService */
private $service;

public function setUp()
{
$this->repository = m::mock('Cribbb\Domain\Model\Identity\UserRepository');

$this->service = new RegisterUserService($this->repository);
}
```

We've already tested the `UserRepository` so there is no need to hit the database during these tests. This is a perfect situation to use a mock.

Next we can write the two tests to check that an exception is thrown:

```
/** @test */
public function should_throw_exception_if_email_is_not_unique()
{
```

```
$user = $this->service->register('name@domain.com', 'username', 'password');
}

/** @test */
public function should_throw_exception_if_username_is_not_unique()
{
$this->setExpectedException('Cribbb\Domain\Model\ValueIsNotUniqueException');

$this->repository->shouldReceive('userOfEmail')->andReturn(true);
$this->repository->shouldReceive('userOfUsername')->andReturn(null);

$user = $this->service->register('name@domain.com', 'username', 'password');
}
```

In each test we simply tell the mock to return `true` to simulate that a user was found with either the email or username and so therefore an exception should be thrown.

Create the User object

Next we can use the `Email`, `Username` and `Password` Value Objects to create a new `User` Entity.

The first thing to do is to use the `UserRepository` to return a new `UserId`:

Next we need to use the `HashingService` to hash the password and return a new instance of `HashedPassword`.

First we need to inject the `HashingService` in through the constructor:

```
/**
* @var UserRepository
*/
private $userRepository;

/**
* @var HashingService
*/
private $hashingService;

/**
* Create a new RegisterUserService
*
* @param UserRepository $userRepository
* @param HashingService $hashingService
* @return void
*/
public function __construct(UserRepository $userRepository, HashingService $hashingService)
{
$this->userRepository = $userRepository;
$this->hashingService = $hashingService;
}
```

We also need to update the test class `setUp()` method:

```
/** @var UserRepository */
private $repository;

/** @var HashingService */
private $hashing;

/** @var RegisterUserService */
private $service;

public function setUp()
{
$this->repository = m::mock('Cribbb\Domain\Model\Identity\UserRepository');
$this->hashing = m::mock('Cribbb\Domain\Services\Identity\HashingService');

$this->service = new RegisterUserService($this->repository, $this->hashing);
}
```

In theory you could just inject the `HashingService` into the `RegisterUserService` as there really isn't anything to mock, however in this instance I'm just going to mock it out.

Next we can pass the `Password` into the `HashingService` and receive a `HashedPassword` in return:

You will notice that this service is making use of two other service classes, but we are in no way relying on the implementation of either of these two classes.

Create the User

Next, we can create a new `User` by passing the `$id`, `$email`, `$username` and `$password` objects to the `register()` method on the `User` object:

```
$user = User::register($id, $email, $username, $password);
```

The `$user` object can now be passed into the `add()` method of the `UserRepository` so it can be persisted to the database:

```
$this->userRepository->add($user);
```

And finally, the `$user` object can be returned from the method:

```
* @param string $email
* @param string $username
* @param string $password
* @return void
*/
public function register($email, $username, $password)
{
$email = new Email($email);
$username = new Username($username);
$password = new Password($password);

$this->checkEmailIsUnique($email);
$this->checkUsernameIsUnique($username);

$id = $this->userRepository->nextIdentity();
$password = $this->hashingService->hash($password);

$user = User::register($id, $email, $username, $password);
$this->userRepository->add($user);

return $user;
}
```

This method will be invoked from an Application Service that will accept the returned `$user` object. This object will be load with Domain Events that were created during registration and will be ready to fire.

```
/** @test */
public function should_register_new_user()
{
$this->repository->shouldReceive('userOfEmail')->andReturn(null);
$this->repository->shouldReceive('userOfUsername')->andReturn(null);
$this->repository->shouldReceive('nextIdentity')->andReturn(UserId::generate());
$this->hashing->shouldReceive('hash')->andReturn(new HashedPassword('password'));
$this->repository->shouldReceive('add');

$user = $this->service->register('name@domain.com', 'username', 'password');
$this->assertInstanceOf('Cribbb\Domain\Model\Identity\User', $user);
}
```

This will ensure all of the internal service methods are called and that an instance of `User` is returned.

Conclusion

We've covered quite a bit in today's article, so I'll try to summarise it all here.

The decision as to whether you should model a particular piece of functionality as a Domain Service or an Application Service is kind of a tricky one to make. It really comes down to the context of the application you are building and what is important to the business you are building it for.

However, if you already have a Domain Object where the functionality would naturally fit, it should always be your first priority to avoid using a Domain Service.

Secondly, you don't have to double test the Repository by allowing Domain Services tests to hit the database. Your tests will be unnecessarily slower with little benefit. In today's article we don't really care about the return values of from the Repository so there is no point in hitting the database.

Now that we have a tested registration process we can be confident to iterate on the functionality to add or modify the on-boarding flow of the application.

And finally, today's article should show the culmination of the work from the last couple of weeks. Instead of lumping all of the responsibility of user registration into a single service, we've move the responsibility into many different Entities, Value Objects, Specifications and Repositories. This means each class has a single responsibility, our code is easier to test and understand and we can reuse that functionality for other tasks without repeating ourselves or coupling code.

This is a series of posts on building an entire Open Source application called Cribbb. All of the tutorials will be free to web, and all of the code is available on GitHub.

To view a full listing of the tutorials in this series, click here.

How to create an Active Record style PHP SDK Part 16

A couple of weeks ago we looked at setting up the foundation for the persistence aspect of this Active Record style PHP SDK. The Active Record pattern states that each model object should have public methods for creating, updating, saving and deleting directly from

PHILIP BROWN
22 OCT 2014 • 6 MIN READ

How to create an Active Record style PHP SDK Part 15

Last week we looked at building out the functionality to serialise model objects into JSON that can be sent to the CapsuleCRM API. We are now at the stage where we can call the toJson() method on a model object and have that object

PHILIP BROWN
15 OCT 2014 • 8 MIN READ