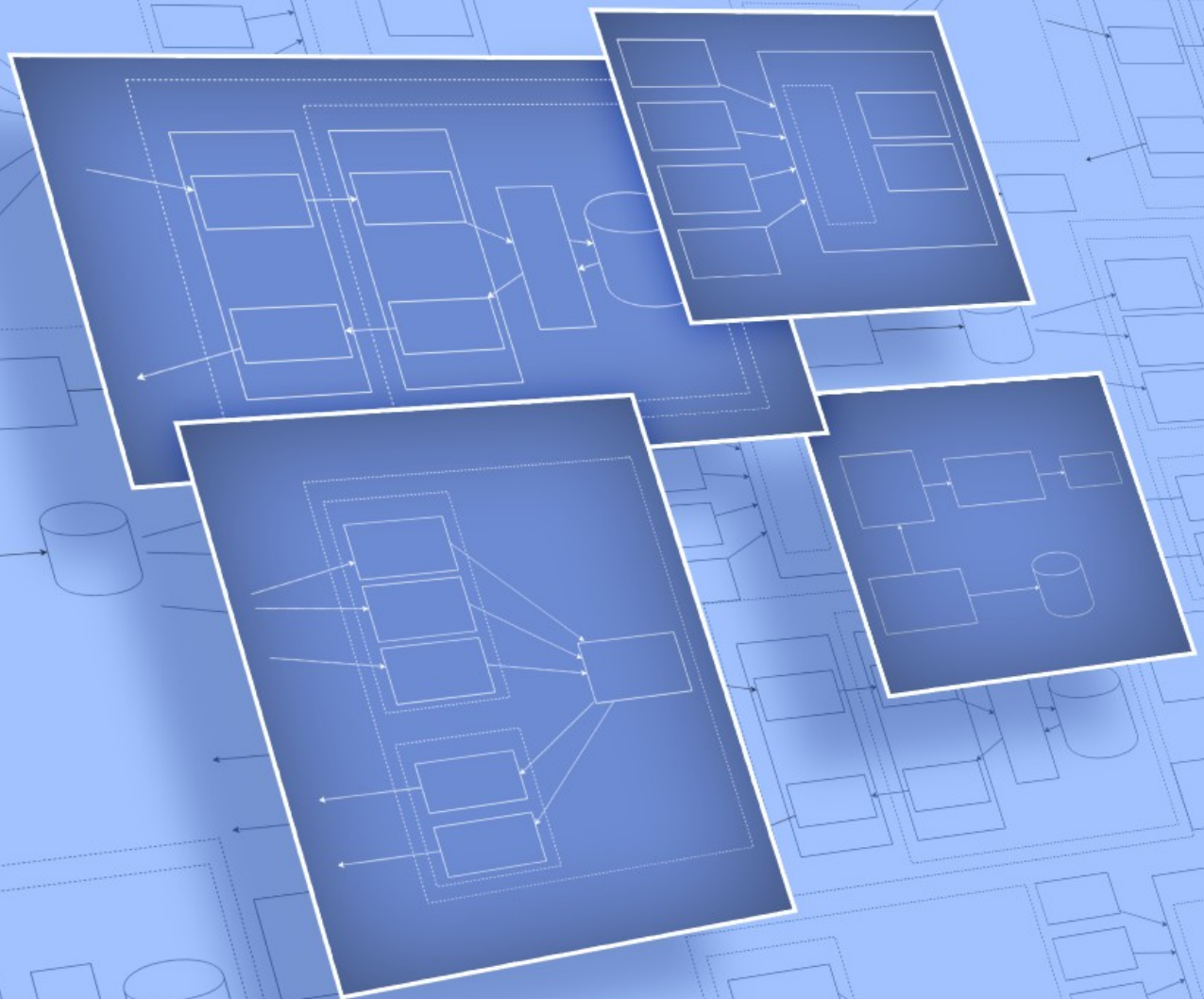
Adel F.

Architecture of complex web applications

With examples in Laravel(PHP)



Architecture of complex web applications

With examples in Laravel(PHP)

Adel F

This book is for sale at http://leanpub.com/architecture-of-complex-web-applications

This version was published on 2019-03-28

* * * * *

* * * * *

Table of Contents

1. Introduction

> “Software Engineering Is Art Of Compromise” wiki.c2.com

I have seen many projects evolve from a simple “MVC” structure. Many developers explain the MVC pattern like this: “View is a blade file, Model is an Eloquent entity, and one Controller to rule them all!” Well, not one, but every additional logic usually implemented in a controller. Controllers become very large containers of very different logic (image uploading, 3rd party APIs calling, working with Eloquent models, etc.). Some logic are moved to base controllers just to reduce the copy-pasting. The same problems exist both in average projects and in big ones with 10+ millions unique visitors per day.

The MVC pattern was introduced in the 1970-s for the Graphical User Interface. Simple CRUD web applications are just an interface between user and database, so the reinvented “MVC for web” pattern became very popular. Although, web applications can easily become more than just an interface for a database. What does the MVC pattern tell us about working with binary files(images, music, videos…), 3rd party APIs, cache? What are we to do if entities have non-CRUD behavior? The answer is simple: Model is not just an Active Record (Eloquent) entity; it contains all logic that work with an application’s data. More than 90 percent of a usual modern complex web application code is Model in terms of MVC. Symfony framework creator Fabien Potencier once said: “I don’t like MVC because that’s not how the web works. Symfony2 is an HTTP framework; it is a Request/Response framework.” The same I can say about Laravel.

Frameworks like Laravel contain a lot of Rapid Application Development features, which help to build applications very fast by allowing developers to cut corners. They are very useful in the “interface for database” phase, but later might become a source of pain. I did a lot of refactorings just to

remove them from the projects. All these auto-magical things and “convenient” validations, like “quickly check that this email is unique in this table” are great, but the developer should fully understand how they work and when better to implement it another way.

On the other hand, advice from cool developers like “your code should be 100% covered by unit tests”, “don’t use static methods”, and “depend only upon abstractions” can become cargo cults for some projects. I saw an interface **IUser** with 50+ properties and class **User: IUser** with all these properties copy-pasted(it was a C# project). I saw a huge amount of “abstractions” just to reach a needed percentage of unit test coverage. These pieces of advice are good, but only for some cases and they must be interpreted correctly. Each best practice contains an explanation of what problem it solves and which conditions the destination project should apply. Developers have to understand them before using them in their projects.

All projects are different. Some of them are suitable for some best practices. Others will be okay without them. Someone very clever said: “Software development is always a trade-off between short term and long term productivity”. If I need some functionality already implemented in the project in another place, I can just copy-paste it. It will be very productive in the short term, but very quickly it will start to create problems. Almost every decision about refactoring or using a pattern has the same dilemma. Sometimes, it is a good decision to not use some pattern that makes the code “better”, because the positive effect for the project will not be worth the amount of time needed to implement it. Balancing between patterns, techniques and technologies and choosing the most suitable combination of them for the current project is one the most important skills of a software developer/architect.

In this book, I talk about common problems appearing in projects and how developers usually solve them. The reasons and conditions for using these solutions are a very important part of the book. I don’t want to create a new cargo cult :)

I have to warn you:

- This book is not for beginners. To understand the problems I will talk about, the developer should at least have participated in one project.
- This book is not a tutorial. Most of the patterns will be observed superficially, just to inform the reader about them and how and when they can help. The links to useful books and articles will be at the end of the book.
- The example code will never be ideal. I can call some code “correct” and still find a lot of problems in it, as shown in the next chapter.

2. Bad Habits

SRP violation

In this chapter, I'll try to show how projects, written according to documentation, usually grow up. How developers of real projects try to fight with complexity. Let's start with a simple example.

```
public function store(Request $request)
{
    $this->validate($request, [
        'email' => 'required|email',
    ]);

    $user = User::create($request->all());

    if(!$user) {
        return redirect()->back()->withMessage('...');
    }

    return redirect()->route('users');
}

public function update($id, Request $request)
{
    //almost the same
}
```

It was very easy to write. Then, new requirements appear. Add avatar uploading in create and update forms. Also, an email should be sent after registration.

```
public function store(Request $request)
{
    $this->validate($request, [
        'email' => 'required|email',
        'avatar' => 'required|image',
    ]);

    $avatarFileName = ...;
    \Storage::disk('s3')->put(
        $avatarFileName, $request->file('avatar'));

    $user = new User($request->except('avatar'));
    $user->avatarUrl = $avatarFileName;
    $user->save();

    \Email::send($user, 'Hi email');
```

```
    return redirect()->route('users');
}
```

Some logic should be copied to **update** method. But, for example, email sending should happen only after user creation. The code still looks okay, but the amount of copy-pasted code grows. Customer asks for a new requirement - automatically check images for adult content. Some developers just add this code(usually, it's just a call for some API) to **store** method and copy-paste it to **update**. Experienced developers extract upload logic to new controller method. More experienced Laravel developers find out that file uploading code becomes too big and instead create a class, for example, **ImageUploader**, where all this upload and adult content-checking logic will be. Also, a Laravel facade(Service Locator pattern implementation) **ImageUploader** will be introduced for easier access to it.

```
/**
 * @returns bool|string
 */
public function upload(UploadedFile $file)
```

This function returns false if something wrong happens, like adult content or S3 error. Otherwise, uploaded image url.

```
public function store(Request $request)
{
    ...
    $avatarFileName = \ImageUploader::upload(
        $request->file('avatar'));

    if($avatarFileName === false) {
        return %some_error%;
    }
    ...
}
```

Controller methods became simpler. All image uploading logic was moved to another class. Good. The project needs image uploading in another place and we already have a class for it! Only one new parameter will be added to **upload** method: a folder where images should be stored will be different for avatars and publication images.

```
public function upload(UploadedFile $file, string $folder)
```

New requirement - immediately ban user who uploaded adult content. Well, it sounds weird, because current image analysis tools aren't very accurate,

but it’s a requirement(it was a real requirement in one of my projects!).

```
public function upload(UploadedFile $file, string $folder)
{
    ...
    if(check failed) {
        $this->banUser(\Auth::user());
    }
    ...
}
```

New requirement - application should not ban user if he uploads something wrong to private places.

```
public function upload(
    UploadedFile $file,
    string $folder,
    bool $dontBan = false)
```

When I say “new requirement”, it doesn’t mean that it appears the next day. In big projects it can take months and years and can be implemented by another developer who doesn’t know why the code is written like this. His job - just implement this task as fast as possible. Even if he doesn’t like some code, it’s hard to estimate how much time this refactoring will take in a big system. And, much more important, it’s hard to not break something. It’s a very common problem. I hope this book will help to organize your code to make it more suitable for safe refactoring. New requirement - user’s private places needs weaker rules for adult content.

```
public function upload(
    UploadedFile $file,
    string $folder,
    bool $dontBan = false,
    bool $weakerRules = false)
```

The last requirement for this example - application shouldn’t ban user immediately. Only after some tries.

```
public function upload(
    UploadedFile $file,
    string $folder,
    bool $dontBan = false,
    bool $weakerRules = false,
    int $banThreshold = 5)
{
    //...
    if(check failed && !$dontBan) {
        if(\RateLimiter::tooManyAttempts(..., $banThreshold)) {
            $this->banUser(\Auth::user());
```

```
        }
    }
    //...
}
```

Okay, this code doesn't look good anymore. Image upload function has a lot of strange parameters about image checking and user banning. If user banning process should be changed, developers have to go to **ImageUploader** class and implement changes there. **upload** function call looks weird:

```
\ImageUploader::upload(
    $request->file('avatar'), 'avatars', true, false);
```

Single Responsibility Principle was violated here. **ImageUploader** class has also some other problems but we will talk about them later. As I mentioned before, store and update methods are almost the same. Let's imagine some very big entity with huge logic and image uploading, other API's calling, etc.

```
public function store(Request $request)
{
    // tons of code
    // especially if some common code haven't
    // extracted to classes like ImageUploader
}

public function update($id, Request $request)
{
    // almost the same tons of code
}
```

Sometimes a developer tries to remove all this copy-paste by extracting the method like this:

```
protected function updateOrCreateSomething(..., boolean $update)
{
    if($update)...
    if($update)...
    if(!$update)...
}
```

I saw this kind of method with 700+ lines. After many requirement changes, there were a huge amount of **if($update)** checks. This is definitely the wrong way to remove copy-pasting. When I refactored this method by creating different **create** and **update** methods and extracting similar logic to their own methods/classes, the code become much easier to read.

CRUD-style thinking

The REST is very popular. Laravel developers use resource controllers with ready store, update, delete, etc. methods even for web routes, not only for API. It looks very simple. Only 4 verbs: **GET**(read), **POST**(create), **PUT/PATCH**(update) and **DELETE**(delete). It is simple when your project is just a CRUD application with create/update forms and lists with a delete button. But when the application becomes a bit more complex, the REST way becomes too hard. For example, I googled “REST API ban user” and the first three results with some API’s documentation were very different.

```
PUT /api/users/banstatus
params:
UserID
IsBanned
Description


POST /api/users/ban userId reason

POST /api/users/un-ban userId


PUT /api/users/{id}/status
params:
status: guest, regular, banned, quarantine

There also was a big table: which status can be changed to which
and what will happen
```

As you see, any non-standard verb(ban) and REST becomes not so simple. Especially for beginners. Usually, all other methods are implemented by the **update** method. When I asked in one of the seminars how to implement user banning with REST, the first answer was:

```
PUT /api/users/{id}
params:
IsBanned=true
```

Ok. IsBanned is the property of User, but when user actually was banned, we should send, for example, an email for this user. This requirement consequences very complicated conditions with comparing “old” and “new” values on user update operation. Another example: password change.

```
PUT /api/users/{id}
params:
oldPassword=***
password=***
```

oldPassword is not a user property. So, another condition at user update. This CRUD-style thinking, as I call it, affects even the user interface. I always remember “typical Apple product, typical Google product” image as the best illustration of the problem.

The worship of PHP dark magic

Sometimes developers just don’t(or don’t want to) see a simple way to implement something. They write code with reflection, magic methods or other PHP dynamic features, code which was hard to write and will be very hard to read! I used to do it regularly. Like everyone, I think. I’ll show a little funny example.

I had a class for cache keys in one of my projects. We need keys in at least 2 places: reading/creating cache value and clearing it before it expires(in cases when entity was changed). Obvious solution:

```
final class CacheKeys
{
    public static function getUserByIdKey(int $id)
    {
        return sprintf('user_%d_%d', $id, User::VERSION);
    }

    public static function getUserByEmailKey(string $email)
    {
        return sprintf('user_email_%s_%d',
            $email,
            User::VERSION);
    }
    //...
}

$key = CacheKeys::getUserByIdKey($id);
```

Do you remember the dogma “Don’t use static functions”? Almost always, it’s true. But this is a good example of exception. We will talk about it in further in the Dependency Injection chapter. Well, when another project needed the same functionality, I showed this class to the developer and said you can do the same. After some time, he said that this class “isn’t very beautiful” and committed this code:

```
/**
 * @method static string getUserByIdKey(int $id)
 * @method static string getUserByEmailKey(string $email)
 */
class CacheKeys
```

```
{
    const USER_BY_ID = 'user_%d';
    const USER_BY_EMAIL = 'user_email_%s';

    public static function __callStatic(
        string $name, array $arguments)
    {
        $cacheString = static::getCacheKeyString($name);
        return call_user_func_array('sprintf',
            array_prepend($arguments, $cacheString));
    }

    protected static function getCacheKeyString(string $input)
    {
        return constant('static::' . static::getConstName($input));
    }

    protected static function getConstName(string $input)
    {
        return strtoupper(
            static::fromCamelCase(
                substr($input, 3, strlen($input) - 6))
        );
    }

    protected static function fromCamelCase(string $input)
    {
        preg_match_all('<huge regexp>', $input, $matches);
        $ret = $matches[0];
        foreach ($ret as &$match) {
            $match = $match == strtoupper($match)
                ? strtolower($match)
                : lcfirst($match);
        }
        return implode('_', $ret);
    }
}

$key = CacheKeys::getUserById($id);
```

Shortly, this code transforms “getUserById” string to “USER_BY_ID” and uses this constant value. A lot of developers, especially young ones, like to make this kind of “beautiful” code. Sometimes, it can save a lot of lines of code. Sometimes not. But it’s always hard to debug and support. The developer should think 10 times before using any “cool” dynamic feature of language.

“Rapid” Application Development

Some framework creators also like dynamic features. In small projects they really help to write quickly. But by using these cool features we are losing control of our app execution and when the project grows, it starts to create problems. In the previous example with cache keys, *::VERSION constants

were forgotten, because it wasn't very simple to use it with this "optimization". Another example; Laravel apps have a lot of the same code like this:

```
class UserController
{
    public function update($id)
    {
        $user = User::find($id);
        if($user === null)
        {
            abort(404);
        }
        //...
    }
}
```

Laravel starting from some version suggests to use implicit route binding. This code does the same as previous:

```
Route::post('api/users/{user}', 'UserController@update');
```

```
class UserController
{
    public function update(User $user)
    {
        //...
    }
}
```

It definitely looks better and reduces a lot of "copy-pasted" code. Later, the project can grow and caching will be implemented. For GET queries, it is better to use cache, but not for POST (there are a lot of reasons to not use cached entities in update operations). Another possible issue: different databases for read and write queries. It happens when one database server can't serve all of a project's queries. Usually, database scaling starts from creating one database to write queries and one or more read databases. Laravel has convenient configurations for read&write databases. So, the route binding code can now look like this:

```
Route::bind('user', function ($id) {
    // get and return cached version or abort(404);
});

Route::bind('userToWrite', function ($id) {
    return App\User::onWriteConnection()->find($id) ?? abort(404);
});

Route::get('api/users/{user}', 'UserController@edit');
Route::post('api/users/{userToWrite}', 'UserController@update');
```

It looks so strange and so easy to make a mistake. It happened because instead of explicitly getting entities by id, the developer used implicit “optimization”. The first example can be shortened like this:

```
class UserController
{
    public function update($id)
    {
        $user = User::findOrFail($id);
        //...
    }
}
```

There is no need to “optimize” this one line of code. Frameworks suggest a lot of other ways to lose control of your code. Be very careful with them.

A few words about Laravel’s convenient configuration for read&write databases. It’s really convenient, but again, we lose control here. It isn’t smart enough. It just uses read connection to select queries and write connection to insert/update/delete queries. Sometimes we need to select from a write connection. It can be solved with ::onWriteConnection() helpers. But, for example, lazy loading relation will be fetched from read connection again! In some very rare cases it made our data inconsistent. Can you imagine how difficult it was to find this bug? In Laravel 5.5, one option was added to fix that. It will send each query to write database after the first write database query. This option partially solves the problem, but looks so weird.

As a conclusion, I can say this: “Less magic in the code - much easier to debug and support it”. Very rarely, in some cases, like ORM, is it okay to make some magic, but only there.

Saving lines of code

When I was studying software engineering in the university, sometimes one of us showed examples of his super-short code. Usually it was one line of code implementing some algorithm. Some days after authoring this code, one could spend a minute or more to understand this one line, but still it was “cool”. My conditions of cool code were changed since those days. Cool code for me is the code with minimum time needed to read and understand it by any other developer. Short code is not always the most

readable code. Usually, several simple classes is much better than one short but complicated class.

The funny real example from one of my projects:

```
public function userBlockage(
    UserBlockageRequest $request, $userId)
{
   /** @var User $user */
   $user = User::findOrFail($userId);

   $done = $user->getIsBlocked()
       ? $user->unblock()
       : $user->block($request->get('reason'));

   return response()->json([
       'status'  => $done,
       'blocked' => $user->getIsBlocked()
   ]);
}
```

The developer wanted to save some lines of code and implemented user blocking and unblocking in the same method. The problems started from naming. The not very precise ‘blockage’ noun instead of the natural ‘block’ and ‘unblock’ verbs. The main problem is concurrency: two moderators could open the same user’s page and try to block him. First one will block, but the other one will unblock! Some kind of optimistic locking could solve this issue, but the idea of optimistic locking not very popular in Laravel projects (I’ve found some packages, but they have less than 50 stars in github). The best solution is to create two separate methods for blocking and unblocking.

Other sources of pain

I forgot to tell you about the main enemy - Copy-Paste Driven Development. I hope the answer for “Why copy-pasted code is hard to support” is obvious. There are lots of other ways to make applications unsupportable and non-extendable. This book is about how to avoid these problems, and the good habits which make the code more flexible and supportable.
Let’s begin!

3. Dependency injection

Single responsibility principle

You may have heard about the Single Responsibility Principle(SRP). It's canonical definition: every module, class, or function should have responsibility over a single part of the functionality provided by the software. Many developers simplify the definition to "program object should do only one thing". This definition is not very precise. Robert C. Martin changes the term "responsibility" to "reason to change": "A class should have only one reason to change". "Reason to change" is a more convenient concept and we can talk about architecture using it. Almost all architectures and best practices are trying to help the code to be more prepared for changes. However, applications are different; they have different requirements and different kinds of changes.

I often read this: "If you place all your code in controllers, it violates SRP!". Imagine an application - simple CRUD(Create, Read, Update and Delete) where users only look and change your data using these 4 operations. All applications are just interfaces for creating, viewing, editing and removing data, and all operations go directly to the database without any other processing.

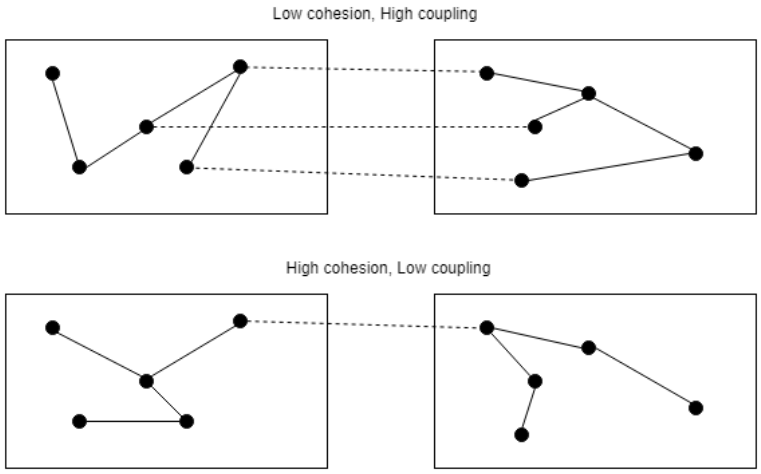
```
public function store(Request $request)
{
    $this->validate($request, [
        'email' => 'required|email',
        'name' => 'required',
    ]);

    $user = User::create($request->all());

    if(!$user) {
        return redirect()->back()->withMessage('...');
    }

    return redirect()->route('users');
}
```

What kind of changes can this app have? Add/remove fields, add/remove entities… It's hard to imagine something more. I don't think this code violates SRP. Almost. Theoretically, redirecting to another route change is possible, but it's not very important. I don't see any reason to refactor this code. New requirements can appear with application growth: user avatar image uploading and email sending:

```
public function store(Request $request)
{
    $this->validate($request, [
        'email' => 'required|email',
        'name' => 'required',
        'avatar' => 'required|image',
    ]);

    $avatarFileName = ...;
    \Storage::disk('s3')->put(
        $avatarFileName, $request->file('avatar'));

    $user = new User($request->except('avatar'));
    $user->avatarUrl = $avatarFileName;

    if(!$user->save()) {
        return redirect()->back()->withMessage('...');
    }

    \Email::send($user->email, 'Hi email');

    return redirect()->route('users');
}
```

Here are several responsibilities already. Something might be changed in image uploading or email sending. Sometimes it's hard to catch the moment when refactoring should be started. If these changes appeared only for user entity, there is probably no sense in changing anything. However, other parts of the application will for sure use the image uploading feature.

I want to talk about two important characteristics of application code - cohesion and coupling. They are very basic and SRP is just a consequence of them. Cohesion is the degree to which all methods of one class or parts of another unit of code (function, module) are concentrated in its main goal. Close to SRP. Coupling between two classes (functions, modules) is the degree of how much they know about each other. High coupling means that some knowledge is shared between several parts of code and each change can cause a cascade of changes in other parts of the application.



Current case with **store** method is a good illustration of losing code quality. It contains several responsibilities - it loses cohesion. Image uploading responsibility is implemented in a few different parts of the application - high coupling. It’s time to extract this responsibility to its own class.

First try:

```
final class ImageUploader
{
    public function uploadAvatar(User $user, UploadedFile $file)
    {
        $avatarFileName = ...;
        \Storage::disk('s3')->put($avatarFileName, $file);

        $user->avatarUrl = $avatarFileName;
    }
}
```

I gave this example because I constantly encounter the fact that developers, trying to take out the infrastructure functionality, take too much with them. In this case, the **ImageUploader** class, in addition to its primary responsibility (file upload), assigns the value to the User class property. What is bad about this? The **ImageUploader** class “knows” about the **User** class and its **avatarUrl** property. Any such knowledge tends to change. You will also have to change the **ImageUploader** class. This is high coupling again.

Lets try to write ImageUploader with a single responsibility:

```php
final class ImageUploader
{
    public function upload(string $fileName, UploadedFile $file)
    {
        \Storage::disk('s3')->put($fileName, $file);
    }
}
```

Yes, this doesn't look like a case where refactoring helped a lot. But let's imagine that ImageUploader also generates a thumbnail or something like that. Even if it doesn't, we extracted its responsibility to its own class and spent very little time on it. All future changes with the image uploading process will be much easier.

Dependency Injection

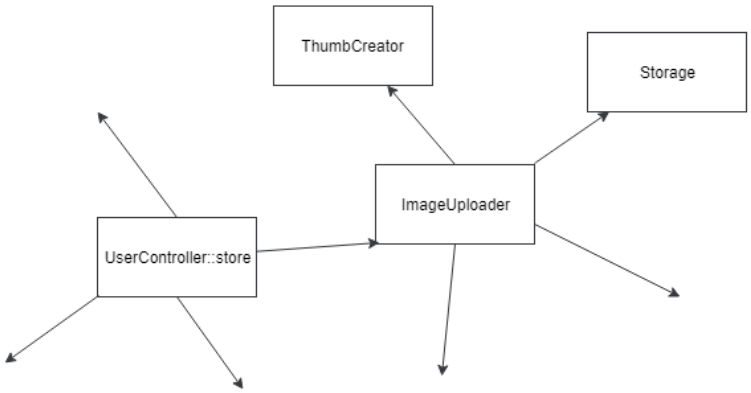
Well, we created **ImageUploader** class, but how do we use it in **UserController::store** method?

```php
$imageUploader = new ImageUploader();
$imageUploader->upload(...);
```

Or just make the **upload** method static and call it like this:

```php
ImageUploader::upload(...);
```

It was easy, right? But now **store** method has a hard-coded dependency to the **ImageUploader** class. Lets imagine a lot of methods with this hard dependency and then the company decided to use another image storage. Not for all images, only for some of them. How would developers usually implement that? They just create **AnotherImageUploader** class and change **ImageUploader** to **AnotherImageUploader** in all needed methods. But what happened? According SRP, each of these methods should have only one reason to change. Why does changing the image storage cause several changes in the **ImageUploader** class dependents?



As you see, the application looks like metal grid. It's very hard to take, for example, the **ImageUploader** class and move it to another project. Or just unit test it. **ImageUploader** can't work without **Storage** and **ThumbCreator** classes, and they can't work without their dependencies, etc. Instead of direct dependencies to classes, the **Dependency Injection** technique suggests just to ask dependencies to be provided to the class.

```
final class ImageUploader
{
    /** @var Storage */
    private $storage;

    /** @var ThumbCreator */
    private $thumbCreator;

    public function __construct(
        Storage $storage, ThumbCreator $thumbCreator)
    {
        $this->storage = $storage;
        $this->thumbCreator = $thumbCreator;
    }

    public function upload(...)
    {
        $this->thumbCreator->...
        $this->storage->...
    }
}
```

Laravel and many other frameworks contain "DI container" - a special service, which takes all responsibilities of creating class instances and injecting them to other classes. So, method store can be rewritten like this:

```
public function store(
    Request $request, ImageUploader $imageUploader)
{
    //...
    $avatarFileName = ...;
    $imageUploader->upload(
        $avatarFileName, $request->file('avatar'));

    //...
}
```

Here, the Laravel feature was used to request dependencies directly in the parameters of the controller method. Dependencies have become softer. Classes do not create dependency instances and do not require static methods. However, both the store method and the **ImageUploader** class refer to specific classes. The Dependency Inversion principle says “High-level modules should not depend on low-level modules. Both should depend on abstractions”. Abstractions should not depend on details. Details should depend on abstractions. The requirement of abstraction in OOP languages is interpreted unequivocally: dependence should be on interfaces, and not on classes. However, I have repeatedly stated that projects are different. Let’s consider two options.

You’ve probably heard about Test-driven Development (TDD) techniques. Roughly speaking, TDD postulates writing tests at the same time as the code. A review of TDD techniques is beyond the scope of this book, so we will look at just one of its faces. Imagine that you need to implement the **ImageUploader** class, but the **Storage** and **ThumbCreator** classes are not yet available. We will discuss unit testing in detail in the corresponding chapter, so we will not dwell on the test code now. You can simply create the **Storage** and **ThumbCreator** interfaces, which are not yet implemented. Then you can simply write the **ImageUploader** class and tests for it, creating mocks from these interfaces (we will talk about mocks later).

```
interface Storage
{
    //...Some methods
}

interface ThumbCreator
{
    //...Some methods
}

final class ImageUploader
{
```

```
    /** @var Storage */
    private $storage;

    /** @var ThumbCreator */
    private $thumbCreator;

    public function __construct(
        Storage $storage, ThumbCreator $thumbCreator)
    {
        $this->storage = $storage;
        $this->thumbCreator = $thumbCreator;
    }

    public function upload(...)
    {
        $this->thumbCreator->...
        $this->storage->...
    }
}

class ImageUploaderTest extends TestCase
{
    public function testSomething()
    {
        $storageMock = \Mockery::mock(Storage::class);
        $thumbCreatorMock = \Mockery::mock(ThumbCreator::class);

        $imageUploader = new ImageUploader(
            $storageMock, $thumbCreatorMock);
        $imageUploader->upload(...
    }
}
```

The **ImageUploader** class still cannot be used in the application, but it has already been written and tested. Later, when the implementations of these interfaces are ready, you can configure the container in Laravel, for example:

```
$this->app->bind(Storage::class, S3Storage::class);
$this->app->bind(ThumbCreator::class, ImagickThumbCreator::class);
```

After that, the **ImageUploader** class can be used in the application. When the container creates an instance of the **ImageUploader** class, it will create instances of the required classes and substitute them instead of interfaces into the constructor. TDD has proven itself in many projects where it is part of the standard. I also like this approach. Developing with TDD, you get little comparable pleasure. However, I have rarely seen its use. It imposes quite serious requirements on the developer for architectural thinking. Developers need to know what to put in separate interfaces and classes and decompose the application in advance.

Usually everything in projects is much simpler. First, the **ImageUploader** class is written, in which the logic of creating thumbnails and the logic of saving everything to the repository are concentrated. Then, perhaps, is the extraction of logic into the classes **Storage** and **ThumbCreator**, leaving only a certain orchestration over these two classes in **ImageUploader**. Interfaces are not used. Occasionally a very interesting event takes place in such projects - one of the developers reads about the Dependency Inversion principle and decides that there are serious problems with the architecture on the project. Classes do not depend on abstractions! Interfaces should be introduced immediately! But the names **ImageUploader**, **Storage**, and **ThumbCreator** are already taken. As a rule, in this situation, developers choose one of two terrible ways to extract the interface.

The first is the creation of *Contracts namespace and the creation of all interfaces there. As an example, Laravel source:

```
namespace Illuminate\Contracts\Cache;

interface Repository
{
    //...
}


namespace Illuminate\Contracts\Config;

interface Repository
{
    //...
}


namespace Illuminate\Cache;

use Illuminate\Contracts\Cache\Repository as CacheContract;

class Repository implements CacheContract
{
    //...
}

namespace Illuminate\Config;

use ArrayAccess;
use Illuminate\Contracts\Config\Repository as ConfigContract;

class Repository implements ArrayAccess, ConfigContract
{
    //...
}
```

There is a double sin here: the use of the same name for the interface and class, as well as the use of the same name for different program objects. The namespace feature provides an opportunity for such detour maneuvers. As you can see, even in the source code of classes, you have to use **CacheContract** and **ConfigContract** aliases. For the rest of the project, we have 4 program objects with the name Repository. And the classes that use the configuration and cache via DI look something like this (if you do not use aliases):

```
use Illuminate\Contracts\Cache\Repository;

class SomeClassWhoWantsConfigAndCache
{
    /** @var Repository */
    private $cache;

    /** @var \Illuminate\Contracts\Config\Repository */
    private $config;

    public function __construct(Repository $cache,
        \Illuminate\Contracts\Config\Repository $config)
    {
        $this->cache = $cache;
        $this->config = $config;
    }
}
```

Only variable names help to guess what dependencies are used here. However, the names for Laravel-facades for these interfaces are quite natural: **Config** and **Cache**. With such names for interfaces, the classes that use them would look much better.

The second option is to use the **Interface** suffix, as such: creating an interface with the name **StorageInterface**. Thus, having class **Storage** implements **StorageInterface**, we postulate that there is an interface and its default implementation. All other classes that implement it, if they exist at all, appear secondary compared to **Storage**. The existence of the **StorageInterface** interface looks very artificial: it was created either to make the code conform to some principles, or only for unit testing. Such a phenomenon is found in many languages. In C#, the **IList** interface and the **List** class, for example. In Java, prefixes or suffixes to interfaces are not accepted, but this often happens there:

```
class StorageImpl implements Storage
```

This is also the situation with the default implementation of the interface. There are two possible situations:

1. There is an interface and several possible implementations. In this case, the interface should be called natural. Implementations should have prefixes that define them. Interface **Storage**. Class **S3Storage** implements **Storage**, class **FileStorage** implements **Storage**.
2. There is an interface and one implementation. Another implementation looks impossible. Then the interface is not needed. It is necessary to use a class with a natural name.

If there is no direct need to use interfaces on the project, then it is quite normal to use classes and Dependency Injection. Let's look again at **ImageUploader**:

```
final class ImageUploader
{
    /** @var Storage */
    private $storage;

    /** @var ThumbCreator */
    private $thumbCreator;

    public function __construct(Storage $storage,
        ThumbCreator $thumbCreator)
    {
        $this->storage = $storage;
        $this->thumbCreator = $thumbCreator;
    }

    public function upload(...)
    {
        $this->thumbCreator->...
        $this->storage->...
    }
}
```

It uses some software objects **Storage** and **ThumbCreator**. The only thing he uses is public methods. It absolutely doesn't care whether it's interfaces or real classes. Dependency Injection, removing the need to instantiate objects from classes, gives us super-abstraction: there is no need for classes to even know what type of program object it is dependent on. At any time, when conditions change, classes can be converted to interfaces with the allocation of functionality to a new class (**S3Storage**). Together with the configuration of the DI-container, these will be the only changes that will have to be made on the project. Of course, if it's a public package, the code

must be written as flexibly as possible and all dependencies must be easily replaceable, therefore interfaces are required. However, on a regular project, using dependencies on real classes is an absolutely normal trade-off.

Inheritance

Inheritance is called one of the main concepts of OOP and developers adore it. However, quite often inheritance is used in the wrong key, when a new class needs some kind of functionality and this class is inherited from the class that has this functionality. A simple example. Laravel has an interface Queue and many classes implementing it. Let's say our project uses **RedisQueue**.

```
interface Queue
{
    public function push($job, $data = '', $queue = null);
}

class RedisQueue implements Queue
{
    public function push($job, $data = '', $queue = null)
    {
        // implementation
    }
}
```

Once it became necessary to log all the tasks in the queue, the result was the **OurRedisQueue** class, which was inherited from **RedisQueue**.

```
class OurRedisQueue extends RedisQueue
{
    public function push($job, $data = '', $queue = null)
    {
        // logging

        return parent::push($job, $data, $queue);
    }
}
```

The task is completed: all **push** methods calls are logged. After some time, the framework is updated and a new method pushOn appears in the **Queue** interface. It is actually a push alias, but with a different order of parameters. The expected implementation appears in the RedisQueue class.

```
interface Queue
{
    public function push($job, $data = '', $queue = null);
    public function pushOn($queue, $job, $data = '');
```

```
}

class RedisQueue implements Queue
{
    public function push($job, $data = '', $queue = null)
    {
        // implementation
    }

    public function pushOn($queue, $job, $data = '')
    {
        return $this->push($job, $data, $queue);
    }
}
```

Because **OurRedisQueue** inherits the **RedisQueue**, we did not need to take any action during the upgrade. Everything works as before and the team gladly began using the new pushOn method.

In the new update, the Laravel team could, for some reason, do some refactoring.

```
class RedisQueue implements Queue
{
    public function push($job, $data = '', $queue = null)
    {
        return $this->innerPush(...);
    }

    public function pushOn($queue, $job, $data = '')
    {
        return $this->innerPush(...);
    }

    public function innerPush(...)
    {
        // implementation
    }
}
```

Refactoring is absolutely natural and doesn’t change the class contract. It still implements the **Queue** interface. However, after some time after this update, the team notices that logging does not always work. It is easy to guess that now it will only log **push** calls, not **pushOn**. When we inherit a non-abstract class, this class has two responsibilities at a high level. A responsibility to their own clients, as well as to the inheritors, who also use its functionality. The authors of the class may not even suspect the second responsibility, and this can lead to complex, elusive bugs on the project. Even such a simple example quite easily led to a bug that would not be so easy to catch. To avoid such difficulties in my projects, all non-abstract

classes are marked as final, thus prohibiting inheritance from myself. The template for creating a new class in my IDE contains the ‘final class’ instead of just the ‘class’. Final classes have responsibility only to their clients.

By the way, Kotlin language designers seem to think the same way and decided to make classes there final by default. If you want your class to be open for inheritance, the ‘open’ or ‘abstract’ keyword should be used:

```
open class Foo {}
```

I like this :)

However, the danger of inheriting an implementation is still possible. An abstract class with protected methods and its descendants can fall into exactly the same situation that I described above. The protected keyword creates an implicit connection between parent class and child class. Changes in the parent can lead to bugs in the children. The DI mechanism gives us a simple and natural opportunity to ask for the implementation we need. The logging issue is easily solved using the **Decorator** pattern:

```
final class LoggingQueue implements Queue
{
    /** @var Queue */
    private $baseQueue;

    /** @var Logger */
    private $logger;

    public function __construct(Queue $baseQueue, Logger $logger)
    {
        $this->baseQueue = $baseQueue;
        $this->logger = $logger;
    }

    public function push($job, $data = '', $queue = null)
    {
        $this->logger->log(...);

        return $this->>baseQueue->push($job, $data, $queue);
    }
}

// configuring container in service provider
$this->app->bind(Queue::class, LoggingQueue::class);

$this->app->when(LoggingQueue::class)
    ->needs(Queue::class)
    ->give(RedisQueue::class);
```

Warning: this code will not work in a real Laravel environment, because the framework has a more complex procedure for initiating these classes. This container configuration will inject an instance of **LoggingQueue** to anyone who wants to get a **Queue**. **LoggingQueue** will get a **RedisQueue** instance as a constructor parameter. The Laravel update with a new **pushOn** method results in an error - **LoggingQueue** does not implement the required method. Thus, we immediately implement logging of this method, also.

Plus, you probably noticed that we now completely control the constructor. In the variant with inheritance, we would have to call parent::__construct and pass on everything that it asks for. This would be an additional, completely unnecessary link between the two classes. As you can see, the decorator class does not have any implicit links between classes and allows you to avoid a whole class of troubles in the future.

Image uploader example

Let's return to the image uploader example from the previous chapter. **ImageUploader** class was extracted from controller to implement the image uploading responsibility. Requirements for this class:

- uploaded image content should be checked for unappropriated content
- if the check is passed, image should be uploaded to some folder
- if the check is failed, user who uploaded this image should be banned after some tries

```
final class ImageUploader
{
    /** @var GoogleVisionClient */
    private $googleVision;

    /** @var FileSystemManager */
    private $fileSystemManager;

    public function __construct(
        GoogleVisionClient $googleVision,
        FileSystemManager $fileSystemManager)
    {
        $this->googleVision = $googleVision;
        $this->fileSystemManager = $fileSystemManager;
    }

    /**
     * @param UploadedFile $file
     * @param string $folder
     * @param bool $dontBan
```

```
     * @param bool $weakerRules
     * @param int $banThreshold
     * @return bool|string
     */
    public function upload(
        UploadedFile $file,
        string $folder,
        bool $dontBan = false,
        bool $weakerRules = false,
        int $banThreshold = 5)
    {
        $fileContent = $file->getContents();

        // Some checking using $this->googleVision,
        // $weakerRules and $fileContent

        if(check failed)
            if(!$dontBan) {
                if(\RateLimiter::..., $banThreshold)) {
                    $this->banUser(\Auth::user());
                }
            }

            return false;
        }

        $fileName = $folder . 'some_unique_file_name.jpg';

        $this->fileSystemManager
            ->disk('...')
            ->put($fileName, $fileContent);

        return $fileName;
    }

    private function banUser(User $user)
    {
        $user->banned = true;
        $user->save();
    }
}
```

Basic refactoring

Simple image uploading responsibility becomes too big and contains some other responsibilities. It definitely needs some refactoring.

If **ImageUploader** will be called from console command, the **Auth::user()** command will return null and **ImageUploader** has to add a ‘!== null’ check to its code. Better to provide the **User** object by another parameter(**User** $uploadedBy), which is always not null. The user banning functionality can be used somewhere else. Now it’s only 2 lines of code, but in the future it may contain some email sending or other actions. Better to create a class for that.

```php
final class BanUserCommand
{
    public function banUser(User $user)
    {
        $user->banned = true;
        $user->save();
    }
}
```

Next, the “ban user after some wrong upload tries” responsibility. **$banThreshold** parameter was added to the function parameters by mistake. It’s constant.

```php
final class WrongImageUploadsListener
{
    const BAN_THRESHOLD = 5;

    /** @var BanUserCommand */
    private $banUserCommand;

    /** @var RateLimiter */
    private $rateLimiter;

    public function __construct(
        BanUserCommand $banUserCommand,
        RateLimiter $rateLimiter)
    {
        $this->banUserCommand = $banUserCommand;
        $this->rateLimiter = $rateLimiter;
    }

    public function handle(User $user)
    {
        $rateLimiterResult = $this->rateLimiter
            ->tooManyAttempts(
                'user_wrong_image_uploads_' . $user->id,
                self::BAN_THRESHOLD);

        if($rateLimiterResult) {
            $this->banUserCommand->banUser($user);
            return false;
        }
    }
}
```

Our system’s wrong image uploading reaction might be changed in the future. These changes will only affect this class. Next, the responsibility to remove is “image content checking”:

```php
final class ImageGuard
{
    /** @var GoogleVisionClient */
    private $googleVision;

    public function __construct(
```

```
        GoogleVisionClient $googleVision)
    {
        $this->googleVision = $googleVision;
    }

    /**
     * @param string $imageContent
     * @param bool $weakerRules
     * @return bool true if content is correct
     */
    public function check(
        string $imageContent,
        bool $weakerRules): bool
    {
        // Some checking using $this->googleVision,
        // $weakerRules and $fileContent
    }
}

final class ImageUploader
{
    /** @var ImageGuard */
    private $imageGuard;

    /** @var FileSystemManager */
    private $fileSystemManager;

    /** @var WrongImageUploadsListener */
    private $listener;

    public function __construct(
        ImageGuard $imageGuard,
        FileSystemManager $fileSystemManager,
        WrongImageUploadsListener $listener)
    {
        $this->imageGuard = $imageGuard;
        $this->fileSystemManager = $fileSystemManager;
        $this->listener = $listener;
    }

    /**
     * @param UploadedFile $file
     * @param User $uploadedBy
     * @param string $folder
     * @param bool $dontBan
     * @param bool $weakerRules
     * @return bool|string
     */
    public function upload(
        UploadedFile $file,
        User $uploadedBy,
        string $folder,
        bool $dontBan = false,
        bool $weakerRules = false)
    {
        $fileContent = $file->getContents();

        if(!$this->imageGuard->check($fileContent, $weakerRules)) {
            if(!$dontBan) {
                $this->listener->handle($uploadedBy);
            }
```

```
            return false;
        }

        $fileName = $folder . 'some_unique_file_name.jpg';

        $this->fileSystemManager
            ->disk('...')
            ->put($fileName, $fileContent);

        return $fileName;
    }
}
```

ImageUploader lost some responsibilities and is happy about it. It doesn't care about how to check images and what will happen with a user who uploaded something wrong now. It only makes some orchestration job. But I still don't like a parameters of upload method. Responsibilities were removed from ImageUploader, but their parameters are still there and upload method calls still look ugly:

```
$imageUploader->upload($file, $user, 'gallery', false, true);
```

Boolean parameters always look ugly and increase the cognitive load for reading the code. The new boolean parameter might be added if the requirement to not check images will appear… I'll try to remove them two different ways:

- OOP way
- Configuration way

OOP way

I'm going to use polymorphism, so I have to introduce interfaces.

```
interface ImageChecker
{
    public function check(string $imageContent): bool;
}

final class StrictImageChecker implements ImageChecker
{
    /** @var ImageGuard */
    private $imageGuard;

    public function __construct(
        ImageGuard $imageGuard)
    {
        $this->imageGuard = $imageGuard;
```

```
    }

    public function check(string $imageContent): bool
    {
        return $this->imageGuard
            ->check($imageContent, false);
    }
}

final class WeakImageChecker implements ImageChecker
{
    /** @var ImageGuard */
    private $imageGuard;

    public function __construct(
        ImageGuard $imageGuard)
    {
        $this->imageGuard = $imageGuard;
    }

    public function check(string $imageContent): bool
    {
        return $this->imageGuard
            ->check($imageContent, true);
    }
}

final class SuperTolerantImageChecker implements ImageChecker
{
    public function check(string $imageContent): bool
    {
        return true;
    }
}
```

ImageChecker interface and three implementations:

- **StrictImageChecker** for checking image content with strict rules
- **WeakImageChecker** for checking image content with weak rules
- **SuperTolerantImageChecker** for cases when image checking is not needed

WrongImageUploadsListener class becomes an interface with 2 implementations:

```
interface WrongImageUploadsListener
{
    public function handle(User $user);
}

final class BanningWrongImageUploadsListener
    implements WrongImageUploadsListener
{
    // implementation is the same.
    // with RateLimiter and BanUserCommand
```

```
}

final class EmptyWrongImageUploadsListener
    implements WrongImageUploadsListener
{
    public function handle(User $user)
    {
        // Just do nothing
    }
}
```

EmptyWrongImageUploadsListener class will be used instead of **$dontBan** parameter.

```
final class ImageUploader
{
    /** @var ImageChecker */
    private $imageChecker;

    /** @var FileSystemManager */
    private $fileSystemManager;

    /** @var WrongImageUploadsListener */
    private $listener;

    public function __construct(
        ImageChecker $imageChecker,
        FileSystemManager $fileSystemManager,
        WrongImageUploadsListener $listener)
    {
        $this->imageChecker = $imageChecker;
        $this->fileSystemManager = $fileSystemManager;
        $this->listener = $listener;
    }

    /**
     * @param UploadedFile $file
     * @param User $uploadedBy
     * @param string $folder
     * @return bool|string
     */
    public function upload(
        UploadedFile $file,
        User $uploadedBy,
        string $folder)
    {
        $fileContent = $file->getContents();

        if (!$this->imageChecker->check($fileContent)) {
            $this->listener->handle($uploadedBy);

            return false;
        }

        $fileName = $folder . 'some_unique_file_name.jpg';

        $this->fileSystemManager
            ->disk('...')
            ->put($fileName, $fileContent);
```

```
        return $fileName;
    }
}
```

The logic of boolean parameters was moved to interfaces and their implementors. Working with **FileSystemManager** also can be simplified by creating a facade for it (I'm talking about **Facade** pattern, not Laravel facades). The only problem now is instantiating the configured **ImageUploader** instance for each client. It can be solved by a combination of Builder and Factory patterns. This will give full control of configuring the needed **ImageUploader** object to client code.

Also, it might be solved by configuring DI-container rules, which **ImageUploader** object will be provided for each client. All configuration will be placed in one container config file. I think for this task the OOP way looks too over-engineered. It might be solved simply by one configuration file.

Configuration way

I'll use a Laravel configuration file to store all needed configuration. config/image.php:

```
return [
    'disk' => 's3',

    'avatars' => [
        'check' => true,
        'ban' => true,
        'folder' => 'avatars',
    ],

    'gallery' => [
        'check' => true,
        'weak' => true,
        'ban' => false,
        'folder' => 'gallery',
    ],
];
```

ImageUploader using Laravel configuration(**Repository** class):

```
final class ImageUploader
{
    /** @var ImageGuard */
    private $imageGuard;
```

```php
    /** @var FileSystemManager */
    private $fileSystemManager;

    /** @var WrongImageUploadsListener */
    private $listener;

    /** @var Repository */
    private $config;

    public function __construct(
        ImageGuard $imageGuard,
        FileSystemManager $fileSystemManager,
        WrongImageUploadsListener $listener,
        Repository $config)
    {
        $this->imageGuard = $imageGuard;
        $this->fileSystemManager = $fileSystemManager;
        $this->listener = $listener;
        $this->config = $config;
    }

    /**
     * @param UploadedFile $file
     * @param User $uploadedBy
     * @param string $type
     * @return bool|string
     */
    public function upload(
        UploadedFile $file,
        User $uploadedBy,
        string $type)
    {
        $fileContent = $file->getContents();

        $options = $this->config->get('image.' . $type);
        if(Arr::get($options, 'check', true)) {

            $weak = Arr::get($options, 'weak', false);

            if(!$this->imageGuard->check($fileContent, $weak)){

                if(Arr::get($options, 'ban', true)) {
                    $this->listener->handle($uploadedBy);
                }

                return false;
            }
        }

        $fileName = $options['folder'] . 'some_unique_file_name.jpg';

        $defaultDisk = $this->config->get('image.disk');

        $this->fileSystemManager
            ->disk(Arr::get($options, 'disk', $defaultDisk))
            ->put($fileName, $fileContent);

        return $fileName;
    }
}
```

Well, the code looks not as clean as the “OOP” variant, but its configuration and implementation are very simple. For the image uploading task I prefer this way, but for other tasks with more complicated configurations or orchestrations, the “OOP” way might be more optimal.

Extending interfaces

Sometimes, we need to extend an interface with some method. In the Domain layer chapter, I’ll need a multiple events dispatch feature in each method of service classes. Laravel’s event dispatcher only has the single dispatch method:

```
interface Dispatcher
{
    //...

    /**
     * Dispatch an event and call the listeners.
     *
     * @param  string|object  $event
     * @param  mixed  $payload
     * @param  bool  $halt
     * @return array|null
     */
    public function dispatch($event,
        $payload = [], $halt = false);
}
```

I need only simple foreach:

```
foreach ($events as $event)
{
    $this->dispatcher->dispatch($event);
}
```

But I don’t want to copy-paste it in each method of each service class. C# and Kotlin language’s “extension method” feature solves this problem:

```
namespace ExtensionMethods
{
    public static class MyExtensions
    {
        public static void MultiDispatch(
            this Dispatcher dispatcher, Event[] events)
        {
            foreach (var event in events) {
                dispatcher.Dispatch(event);
            }
        }
```

```
    }
}
```

Then, each class can use MultiDispatch method:

```
using ExtensionMethods;

//...

dispatcher.MultiDispatch(events);
```

PHP doesn’t have this feature. For your own interfaces, the new method can be added to the interface and implemented in each implementor. In case of an abstract class(instead of interface), the method can be added right there without touching inheritors. That’s why I usually prefer abstract classes. For vendor’s interfaces, this is not possible, so the usual solution is:

```
use Illuminate\Contracts\Events\Dispatcher;

abstract class BaseService
{
    /** @var Dispatcher */
    private $dispatcher;

    public function __construct(Dispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    protected function dispatchEvents(array $events)
    {
        foreach ($events as $event)
        {
            $this->dispatcher->dispatch($event);
        }
    }
}

final class SomeService extends BaseService
{
    public function __construct(..., Dispatcher $dispatcher)
    {
        parent::__construct($dispatcher);
        //...
    }

    public function someMethod()
    {
        //...

        $this->dispatchEvents($events);
    }
}
```

Using inheritance just to extend functionality is not a good idea. Constructors become more complicated with **parent::** calls. Extending another interface will consequence changing all constructors.

Creating a new interface is a more natural solution. Service classes need only one **multiDispatch** method from the dispatcher, so I can just make a new interface:

```
interface MultiDispatcher
{
    public function multiDispatch(array $events);
}
```

and implement it:

```
use Illuminate\Contracts\Events\Dispatcher;

final class LaravelMultiDispatcher implements MultiDispatcher
{
    /** @var Dispatcher */
    private $dispatcher;

    public function __construct(Dispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function multiDispatch(array $events)
    {
        foreach($events as $event)
        {
            $this->dispatcher->dispatch($event);
        }
    }
}

class AppServiceProvider extends ServiceProvider
{
    public function boot()
    {
        $this->app->bind(
            MultiDispatcher::class,
            LaravelMultiDispatcher::class);
    }
}
```

BaseService class can be deleted, and service classes will just use this new interface:

```
final class SomeService
{
    /** @var MultiDispatcher */
    private $dispatcher;
```

```
    public function __construct(..., MultiDispatcher $dispatcher)
    {
        //...
        $this->dispatcher = $dispatcher;
    }

    public function someMethod()
    {
        //...

        $this->dispatcher->multiDispatch($events);
    }
}
```

As a bonus, now I can switch from the Laravel events engine to another, just by another implementation of the **MultiDispatcher** interface.

When clients want to use the full interface, just with a new method, a new interface can extend the base one:

```
interface MultiDispatcher extends Dispatcher
{
    public function multiDispatch(array $events);
}

final class LaravelMultiDispatcher
    implements MultiDispatcher
{
    /** @var Dispatcher */
    private $dispatcher;

    public function __construct(Dispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function multiDispatch(array $events)
    {
        foreach($events as $event)
        {
            $this->dispatcher->dispatch($event);
        }
    }

    public function listen($events, $listener)
    {
        $this->dispatcher->listen($events, $listener);
    }

    public function dispatch(
        $event, $payload = [], $halt = false)
    {
        $this->dispatcher->dispatch($event, $payload, $halt);
    }
```

```
    // Other Dispatcher methods
}
```

For big interfaces, it might be annoying to delegate each method there. Some IDEs for other languages(like C#) have commands to do it automatically. I hope PHP IDEs will implement that, too.

Traits

PHP traits are the magical way to “inject” dependencies for free. They are very powerful: they can access private fields of the main class and add new public and even private methods there. I don’t like them, because they are part of PHP dark magic, powerful and dangerous. I use them in unit test classes, because there is no good reason to implement the Dependency Injection pattern there, but avoid doing it in main application code. Traits are not OOP, so every case with them can be implemented using OOP.

Traits extend interfaces

Multi dispatcher issue can be solved with traits:

```
trait MultiDispatch
{
    public function multiDispatch(array $events)
    {
        foreach($events as $event)
        {
            $this->dispatcher->dispatch($event);
        }
    }
}

final class SomeService
{
    use MultiDispatch;

    /** @var Dispatcher */
    private $dispatcher;

    public function __construct(..., Dispatcher $dispatcher)
    {
        //...
        $this->dispatcher = $dispatcher;
    }

    public function someMethod()
    {
        //...

        $this->multiDispatch($events);
```

```
    }
}
```

The **MultiDispatch** trait assumes that the host class has a dispatcher field of the **Dispatcher** class. It is better to not make these kinds of implicit dependencies. A solution with the **MultiDispatcher** interface is more convenient and explicit.

Traits as partial classes

C# language has the partial classes feature. It can be used when a class becomes too big and the developer wants to separate it for different files:

```
// Foo.cs file
partial class Foo
{
    public void bar(){}
}

// Foo2.cs file
partial class Foo
{
    public void bar2(){}
}

var foo = new Foo();
foo.bar();
foo.bar2();
```

When the same happens in PHP, traits can be used as a partial class. Example from Laravel:

```
class Request extends SymfonyRequest
    implements Arrayable, ArrayAccess
{
    use Concerns\InteractsWithContentTypes,
        Concerns\InteractsWithFlashData,
        Concerns\InteractsWithInput,
```

Big **Request** class has been separated for several traits. When some class “wants” to be separated, it’s a very big hint: this class has too many responsibilities. **Request** class can be **composed** by **Session**, **RequestInput** and other classes. Instead of combining a class with traits, it is better to separate the responsibilities, create a class for each of them, and use composition to use them together. Actually, the constructor of **Request** class tells a lot:

```
class Request
{
    public function __construct(
        array $query = array(),
        array $request = array(),
        array $attributes = array(),
        array $cookies = array(),
        array $files = array(),
        array $server = array(),
        $content = null)
    {
        //...
    }

    //...
}
```

Traits as a behavior

Eloquent traits, such as **SoftDeletes**, are examples of behavior traits. They change the behavior of classes. Eloquent classes contain at least two responsibilities: storing entity state and fetching/saving/deleting entities from a database, so behavior traits can also change the way entities are fetched/saved/deleted and add new fields and relations there. What about the configuration of a trait? There are a lot of possibilities. **SoftDeletes** trait:

```
trait SoftDeletes
{
    /**
     * Get the name of the "deleted at" column.
     *
     * @return string
     */
    public function getDeletedAtColumn()
    {
        return defined('static::DELETED_AT')
            ? static::DELETED_AT
            : 'deleted_at';
    }
}
```

It searches the ‘DELETED_AT’ constant in the class. If there is no such constant, it uses the default value. Even for this simple configuration, traits have to use magic (defined function). Other Eloquent traits have more complicated configurations. I’ve found one library and trait that has several configuration variables and methods. It looks like:

```
trait DetectsChanges
{
    //...
    public function shouldLogUnguarded(): bool
```

```
    {
        if (! isset(static::$logUnguarded)) {
            return false;
        }
        if (! static::$logUnguarded) {
            return false;
        }
        if (in_array('*', $this->getGuarded())) {
            return false;
        }
        return true;
    }
}
```

The same magic but with ‘isset’…

Just imagine:

```
class SomeModel
{
    protected function behaviors(): array
    {
        return [
            new SoftDeletes('another_deleted_at'),
            DetectsChanges::create('column1', 'column2')
                ->onlyDirty()
                ->logUnguarded()
        ];
    }
}
```

Explicit behaviors with a convenient configuration without polluting the host class. Excellent! Fields and relations in Eloquent are virtual, so its implementations are also possible.

Traits can also add public methods to the host class interface… I don’t think it’s a good idea, but it’s also possible with something like macros, which are widely used in Laravel. Active record implementations are impossible without magic, so traits and behaviors will also contain it, but behaviors look more explicit, more object oriented, and configuring them is much easier.

Of course, Eloquent behaviors exist only in my imagination. I tried to imagine a better alternative and maybe I don’t understand some possible problems, but I definitely like them more than traits.

Useless traits

Some traits are just useless. I found this one in Laravel sources:

```
trait DispatchesJobs
{
    protected function dispatch($job)
    {
        return app(Dispatcher::class)->dispatch($job);
    }

    public function dispatchNow($job)
    {
        return app(Dispatcher::class)->dispatchNow($job);
    }
}
```

I don’t know why one method is protected and another one is public… I think it’s just a mistake. It just adds the methods from **Dispatcher** to the host class.

```
class WantsToDispatchJobs
{
    use DispatchesJobs;

    public function someMethod()
    {
        //...

        $this->dispatch(...);
    }
}
```

Accessing this functionality without a trait is simpler in Laravel:

```
class WantsToDispatchJobs
{
    public function someMethod()
    {
        //...

        \Bus::dispatch(...);

        //or just

        dispatch(...);
    }
}
```

This “simplicity” is the main reason why people don’t use Dependency Injection in PHP.

```
class WantsToDispatchJobs
{
    /** @var Dispatcher */
```

```
    private $dispatcher;

    public function __construct(Dispatcher $dispatcher)
    {
        $this->dispatcher = $dispatcher;
    }

    public function someMethod()
    {
        //...

        $this->dispatcher->dispatch(...);
    }
}
```

This class is much simpler than previous examples, because the dependency on **Dispatcher** is explicit, not implicit. It can be used in any application, which can create the **Dispatcher** instance. It doesn't need a Laravel facade, trait or 'dispatch' function. The only problem is bulky syntax with the constructor and private field. Even with convenient auto-completion from the IDE, it looks a bit noisy. Kotlin language syntax is much more elegant:

```
class WantsToDispatchJobs(val dispatcher: Dispatcher)
{
    //somewhere...
    dispatcher.dispatch(...);
}
```

PHP syntax is a big barrier to using DI. I hope something will reduce it in the future (language syntax or IDE improvements).

After years of using and not using traits, I can say that developers create traits for two reasons:

- to patch architectural problems
- to create architectural problems

It's much better to fix the problems instead of patching them.

Static methods

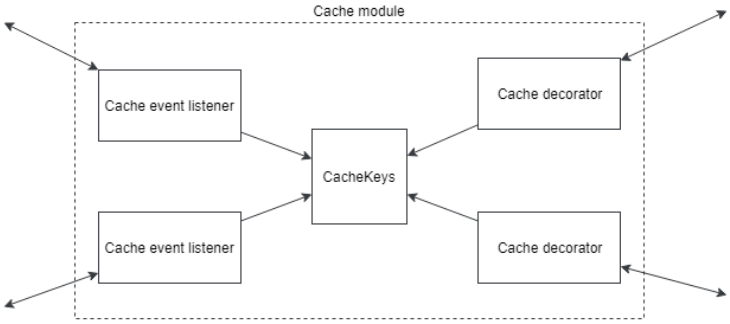
I wrote that using a static method of another class creates a hard coded dependency, but sometimes it's okay. Example from previous chapter:

```
final class CacheKeys
{
    public static function getUserByIdKey(int $id)
```

```
    {
        return sprintf('user_%d_%d', $id, User::VERSION);
    }

    public static function getUserByEmailKey(string $email)
    {
        return sprintf('user_email_%s_%d',
            $email,
            User::VERSION);
    }
    //...
}

$key = CacheKeys::getUserByIdKey($id);
```

Cache keys are needed in at least two places: cache decorators for data fetching classes and event listeners to catch entity changed events, and delete old ones from the cache.

I could use this **CacheKeys** class by DI, but it doesn't make sense. All these decorator and listener classes form some structure which can be called a "cache module" for this app. **CacheKeys** class will be a private part of this module. All other application code shouldn't know about it.



Using static methods for these kinds of internal dependencies that don't work with the outside world(files, database, APIs) is normal practice.

Conclusion

One of the biggest advantages of using the Dependency Injection technique is the explicit contract of each class. The public methods of this class fully describe what it can do. The constructor parameters fully describe what this class needs to do its job.

In big, long-term projects it’s a big advantage: classes can be easily unit tested and used in different conditions(with dependencies provided). All these magic methods, like **__call**, Laravel facades and traits break this harmony.

However, I can’t imagine HTTP controllers outside Laravel applications and almost nobody unit tests them. That’s why I use typical helper functions (**redirect()**, **view()**) and Laravel facades (**Response**, **URL**) there.

4. Painless refactoring

“Static” typing

Big, long-term projects should be regularly refactored to be in good shape. Extracting methods and classes from another method or class. Renaming them, adding or removing parameters. Switching from one (for example, deprecated) method or class to another. Modern IDEs have a lot of tools that make refactoring easy, sometimes fully automatic. However, it might not be easy in PHP.

```
public function publish($id)
{
    $post = Post::find($id);
    $post->publish();
}

// Or

public function makePublish($post)
{
    $post->publish();
}
```

In both of these cases, the IDE can’t find out that **Post::publish** method was called. Let’s try to add a parameter to this method.

```
public function publish(User $publisher)
```

Next, we have to find all publish method calls and add a publisher there. The IDE can’t automatically find it, because of dynamic typing in PHP. So, we have to find all ‘publish’ words in the whole project and filter it(there can be a lot of other ‘publish’ methods or comments).

Next, let’s imagine the team found a non-email string in the email field of the database. How did it happen? We need to find all **User::$email** field usages. But how? The email field doesn’t even exist. It is virtual. We can try to find all ‘email’ words in our project, but it can be set like this:

```
$user = User::create($request->all());
//or
```

```
$user->fill($request->all())
```

All this automagic can sometimes make for big, unpleasant surprises. These bugs in production should be found as soon as possible. Every minute counts.

After some hard refactorings and debugging, a new rule appeared in my projects: make PHP as static as possible. The IDE should understand everything about each method and field. Parameters type hinting and return type declaration should be used with full power. When it's not enough, phpDoc should help.

```
public function makePublish(Post $post)
{
    $post->publish();
}
```

Features, like implicit binding in Laravel that allow one to get models automatically can help, but remember about the danger I mentioned earlier!

```
public function publish(Post $post)
{
    $post->publish();
}

// Or, with phpDoc

public function publish($id)
{
    /**
     * @var Post $post
     */
    $post = Post::find($id);
    $post->publish();
}
```

phpDoc can help in some complex cases:

```
/**
 * @var Post[] $posts
 */
$posts = Post::all();
foreach($posts as $post)
{
    $post->// Here IDE should autocomplete
           // all methods and fields of Post class
}
```

Autocomplete is very convenient when you write code, but it's even better. If IDE autocompletes your code, it understands where these methods and fields are from and it will find them when we ask.

If a function can return an object of some class, it should be declared as a return type directly (starting from PHP 7) or in the **@return** tag of the function's phpDoc:

```
public function getPost($id): Post
{
    //...
}

/**
 * @return Post[] | Collection
 */
public function getPostsBySomeCriteria(...)
{
    return Post::where(...)->get();
}
```

While discussing that, I've been asked a lot of questions, like: "Why are you making Java from PHP?" I'm not making Java from PHP, I'm just making very small comments or type hinting just to have autocomplete now and a huge help with refactoring and debugging in the future. Even for smaller projects it might be super useful.

Templates

Nowadays, more and more projects are becoming API-only. Still, some of them use template engines to generate HTML. There are a lot of method and field calls, too. A usual view call in Laravel:

```
return view('posts.create', [
    'author' => \Auth::user(),
    'categories' => Category::all(),
]);
```

It looks like a function call. Compare with this pseudo-code:

```
/**
* @param User $author
* @param Category[] | Collection $categories
 */
function showPostCreateView(User $author, $categories): string
{
    //
```

```
}

return showPostCreateView(\Auth::user(), Category::all());
```

So, we need the view parameters to be described. It's easy when templates are php files - phpDoc works as usual. But it is not easy for template engines like Blade and depends on the IDE. I use PhpStorm, so can only talk about this IDE. For Blade templates, phpDoc support is also implemented:

```
<?php
/**
 * @var \App\Models\User $author
 * @var \App\Models\Category[] $categories
 */
?>

@foreach($categories as $category)
    {{$category->//Category class fields and methods autocomplete}}
@endforeach
```

I know, it may look very weird and seem like a useless waste of time for you. But after all of this static typing, my code becomes much more flexible. I can, for example, automatically rename methods. Each refactoring brings minimal pain.

Model fields

Using of php magic methods(**__get**, **__set**, **__call**, etc.) is highly discouraged. It will be hard to find all their usages in the project. If you use them, phpDocs should be added to the class. For example, a little Eloquent model class:

```
class User extends Model
{
    public function roles()
    {
        return $this->hasMany(Role::class);
    }
}
```

This class has virtual fields, based on fields of 'users' table and also the 'roles' virtual field as a relationship. There are lot of tools which help with that. For Laravel Eloquent, I use the laravel-ide-helper package. Just one command in console and it generates super useful phpDocs for each Eloquent class:

```php
/**
 * App\User
 *
 * @property int $id
 * @property string $name
 * @property string $email
 * @property-read Collection|\App\Role[] $roles
 * @method static Builder|\App\User whereEmail($value)
 * @method static Builder|\App\User whereId($value)
 * @method static Builder|\App\User whereName($value)
 * @mixin \Eloquent
 */
class User extends Model
{
    public function roles()
    {
        return $this->hasMany(Role::class);
    }
}

$user = new User();
$user->// Here IDE will autocomplete fields!
```

This command should be run every time after the database or model relations change.

Let’s return to our example:

```php
public function store(Request $request, ImageUploader $imageUploader)
{
    $this->validate($request, [
        'email' => 'required|email',
        'name' => 'required',
        'avatar' => 'required|image',
    ]);

    $avatarFileName = ...;
    $imageUploader->upload($avatarFileName, $request->file('avatar'));

    $user = new User($request->except('avatar'));
    $user->avatarUrl = $avatarFileName;

    if(!$user->save()) {
        return redirect()->back()->withMessage('...');
    }

    \Email::send($user->email, 'Hi email');

    return redirect()->route('users');
}
```

User entity creation looks a bit weird. Before, it was like this:

```php
User::create($request->all());
```

Then we had to change it, because the avatar field should not be used directly.

```
$user = new User($request->except('avatar'));
$user->avatarUrl = $avatarFileName;
```

It’s not only looking weird, but also vulnerable. This store method is usual for user registration. Then, for example, the ‘admin’ field will be added to the ‘users’ table. It will be accessible in the admin area to change to each user. But a hacker can just add something like this to the register form:

```
<input type="hidden" name="admin" value="1">
```

And he will be an admin right after registration! For these reasons, some experts suggest to use:

```
$request->only(['email', 'name']);
```

But, if we have to list all of them, it may be better to set the values to name and email fields directly:

```
$user = new User();
$user->email = $request['email'];
$user->name = $request['name'];
$user->avatarUrl = $avatarFileName;
```

Now IDE can find this **email** property using.

“What if I have 50 fields?” First, maybe the UI should be changed :) 50 fields is too many. If not, later in this book I’ll show how fields can be composed to something more suitable and convenient.

Well, we made our code more convenient to refactoring and debugging. This “static typing” isn’t required, but it is so helpful. You should at least try it.

5. Application layer

Let’s continue with our example as the application grows. New fields are added to the registration form: birth date and a checkbox about subscribing to our newsletters.

```
public function store(
    Request $request,
    ImageUploader $imageUploader)
{
    $this->validate($request, [
        'email' => 'required|email',
        'name' => 'required',
        'avatar' => 'required|image',
        'birthDate' => 'required|date',
    ]);

    $avatarFileName = ...;
    $imageUploader->upload(
        $avatarFileName, $request->file('avatar'));

    $user = new User();
    $user->email = $request['email'];
    $user->name = $request['name'];
    $user->avatarUrl = $avatarFileName;
    $user->subscribed = $request->has('subscribed');
    $user->birthDate = new DateTime($request['birthDate']);

    if(!$user->save()) {
        return redirect()->back()->withMessage('...');
    }

    \Email::send($user->email, 'Hi email');

    return redirect()->route('users');
}
```

The application continues growing. Some API appears and user registration should be implemented there, too. Also, some users importing from console command should be implemented. And a Facebook bot! Users have to register there, too. As you see, there are lots of interfaces that want to make some actions with our app. Not only user registration, but almost all other actions that are accessible in the traditional web interface. The most natural solution here is to extract the common logic with some entity (**User**, in this example) to a new class. This kind of class is usually called a “service class”:

```
final class UserService
{
    public function getById(...): User;
    public function getByEmail(...): User;

    public function create(...);
    public function ban(...);
    ...
}
```

But multiple interfaces (API, Web, etc.) are not the only reason to extract service classes. Action methods can grow, and usually there will be two big parts: the business logic and web logic. Pseudo-example:

```
public function doSomething(Request $request, $id)
{
    $entity = Entity::find($id);

    if(!$entity) {
        abort(404);
    }

    if(count($request['options']) < 2) {
        return redirect()->back()->withMessage('...');
    }

    if($entity->something) {
        return redirect()->back()->withMessage('...');
    }

    \Db::transaction(function () use ($request, $entity) {
        $entity->someProperty = $request['someProperty'];

        foreach($request['options'] as $option) {
            //...
        }

        $entity->save();
    });

    return redirect()->...
}
```

This method has at least 2 different responsibilities - Http request/response processing and business logic. Each time the developer changes the http part, he has to read a lot of business code and vice versa. This code is hard to debug and maintain. Refactoring is also difficult. So, extracting business logic to service classes is a good solution here, too.

Request data passing

Lets start to create our **UserService**. The first problem is how to provide data there. Some actions don't need much data, like the remove post action needing only the post id as a parameter, but the user create action needs a lot. We can't use web **Request** class; it's accessible only in web interfaces. So, let's try simple arrays:

```
final class UserService
{
    /** @var ImageUploader */
    private $imageUploader;

    /** @var EmailSender */
    private $emailSender;

    public function __construct(
        ImageUploader $imageUploader, EmailSender $emailSender)
    {
        $this->imageUploader = $imageUploader;
        $this->emailSender = $emailSender;
    }

    public function create(array $request)
    {
        $avatarFileName = ...;
        $this->imageUploader->upload(
            $avatarFileName, $request['avatar']);

        $user = new User();
        $user->email = $request['email'];
        $user->name = $request['name'];
        $user->avatarUrl = $avatarFileName;
        $user->subscribed = isset($request['subscribed']);
        $user->birthDate = new DateTime($request['birthDate']);

        if(!$user->save()) {
            return false;
        }

        $this->emailSender->send($user->email, 'Hi email');

        return true;
    }
}

public function store(Request $request, UserService $userService)
{
    $this->validate($request, [
        'email' => 'required|email',
        'name' => 'required',
        'avatar' => 'required|image',
        'birthDate' => 'required|date',
    ]);

    if(!$userService->create($request->all())) {
        return redirect()->back()->withMessage('...');
    }
```

```
    return redirect()->route('users');
}
```

I just extracted logic without any other refactorings and here I see some problems. When we try to use it from console, for example, there will be code like this:

```
$data = [
    'email' => $email,
    'name' => $name,
    'avatar' => $avatarFile,
    'birthDate' => $birthDate->format('Y-m-d'),
];

if($subscribed) {
    $data['subscribed'] = true;
}

$userService->create($data);
```

Looks a bit weird. By extracting request data, we moved HTML forms logic to our service class. Boolean field values are checked by its existence in the array. Datetime fields(actually, all field types) are parsed from strings. When we try to use this logic in another environment (API, Bots, Console) it becomes very inconvenient. We need another way to provide data to service classes. The most common pattern to transfer data between layers is **Data Transfer Object**(DTO).

```
final class UserCreateDto
{
    /** @var string */
    private $email;

    /** @var DateTime */
    private $birthDate;

    /** @var bool */
    private $subscribed;

    public function __construct(
        string $email, DateTime $birthDate, bool $subscribed)
    {
        $this->email = $email;
        $this->birthDate = $birthDate;
        $this->subscribed = $subscribed;
    }

    public function getEmail(): string
    {
        return $this->email;
    }

    public function getBirthDate(): DateTime
```

```
    {
        return $this->birthDate;
    }

    public function isSubscribed(): bool
    {
        return $this->subscribed;
    }
}
```

Very often I hear something like: “I don’t want to create a whole class just to provide data. Arrays are okay!” Maybe it’s true, but to create a class like UserCreateDto in a modern IDE like PhpStorm is: type the name of it in the “Create class” dialog, hot key to create a constructor(Alt+Ins > Constructor…), type fields with type hinting in constructor parameters, hot key to create fields from constructor parameters and fill them with values(Alt+Enter on constructor parameters > Initialize fields) and then hot key to create getters automatically (Alt+Ins in class body > Getters…). Less than 30 seconds and we have a convenient class with full type hinting.

```
final class UserService
{
    //...

    public function create(UserCreateDto $request)
    {
        $avatarFileName = ...;
        $this->imageUploader->upload(
            $avatarFileName, $request->getAvatarFile());

        $user = new User();
        $user->email = $request->getEmail();
        $user->name = $request->getName();
        $user->avatarUrl = $avatarFileName;
        $user->subscribed = $request->isSubscribed();
        $user->birthDate = $request->getBirthDate();

        if(!$user->save()) {
            return false;
        }

        $this->emailSender->send($user->email, 'Hi email');

        return true;
    }
}

public function store(Request $request, UserService $userService)
{
    $this->validate($request, [
        'email' => 'required|email',
        'name' => 'required',
        'avatar' => 'required|image',
        'birthDate' => 'required|date',
```

```
    ]);

    $dto = new UserCreateDto(
        $request['email'],
        new DateTime($request['birthDate']),
        $request->has('subscribed'));

    if(!$userService->create($dto)) {
        return redirect()->back()->withMessage('...');
    }

    return redirect()->route('users');
}
```

Now it looks canonical. The service class gets pure DTO and executes the action. But the controller action code looks too noisy. **UserCreateDto** constructor can be very big, and maybe we will have to use the Builder pattern or just make the fields public instead of private.

The request validation and DTO object creation can be moved to some class which will consolidate these actions. Laravel has a suitable form request class for that:

```
final class UserCreateRequest extends FormRequest
{
    public function rules()
    {
        return [
            'email' => 'required|email',
            'name' => 'required',
            'avatar' => 'required|image',
            'birthDate' => 'required|date',
        ];
    }

    public function authorize()
    {
        return true;
    }

    public function getDto(): UserCreateDto
    {
        return new UserCreateDto(
            $this->get('email'),
            new DateTime($this->get('birthDate')),
            $this->has('subscribed'));
    }
}

final class UserController extends Controller
{
    public function store(
        UserCreateRequest $request, UserService $userService)
    {
        if(!$userService->create($request->getDto())) {
```

```
            return redirect()->back()->withMessage('...');
        }

        return redirect()->route('users');
    }
}
```

If some class asks **FormRequest** class as a dependency, Laravel creates it and makes a validation automatically. In case of invalid data **store** action won't be executed, and that's why **store** method can be sure that data in **UserCreateRequest** object is always valid.

The Laravel form request class has a **rules** method to validate request data and an **authorize** method to authorize the request. Form request is definitely the wrong place to authorize itself. Authorization usually requires some context, like which entity by which user… It can be retrieved in the authorize method, but it is better to do it in the service class. There will be all needed data. So, it is better to create a base **FormRequest** class with the authorize method returning true and then forget about this method.

Work with database

Let me start from a simple example:

```
class PostController
{
    public function publish($id, PostService $postService)
    {
        $post = Post::find($id);

        if(!$post) {
            abort(404);
        }

        if(!$postService->publish($post)) {
            return redirect()->back()->withMessage('...');
        }

        return redirect()->route('posts');
    }
}

final class PostService
{
    public function publish(Post $post)
    {
        $post->published = true;
        return $post->save();
    }
}
```

Publishing post is one of the simplest examples of a non-CRUD action and I'll use it a lot. Everything seems okay, but when we try to use it from another interface, like console, we will have to implement entity fetching from the database again.

```
public function handle(PostService $postService)
{
    $post = Post::find(...);

    if(!$post) {
        $this->error(...);
        return;
    }

    if(!$postService->publish($post)) {
        $this->error(...);
    } else {
        $this->info(...);
    }
}
```

It's an example of violating the Single Responsibility principle and high coupling. Every part of the application works with the database. Any database-related change will result in changes to the whole app. Sometimes I see strange solutions using services getById(or just get) method:

```
class PostController
{
    public function publish($id, PostService $postService)
    {
        $post = $postService->getById($id);

        if(!$postService->publish($post)) {
            return redirect()->back()->withMessage('...');
        }

        return redirect()->route('posts');
    }
}
```

This code gets the post entity but everything it does with this entity just providing it to another service method. Controller action **publish** doesn't need an entity. It just should ask service to publish this post. The most simple and logical way is:

```
class PostController
{
    public function publish($id, PostService $postService)
    {
        if(!$postService->publish($id)) {
            return redirect()->back()->withMessage('...');
```

```
        }

        return redirect()->route('posts');
    }
}

final class PostService
{
    public function publish(int $id)
    {
        $post = Post::find($id);

        if(!$post) {
            return false;
        }

        $post->published = true;

        return $post->save();
    }
}
```

One of the most important pro's of extracting service classes is consolidating work with business layers and infrastructure, including storages like database and files, in one place and leaving the Web, API, Console and other interfaces to work only with their own responsibilities. The Web part should only prepare requests to the service class and show the results to user. The same is true for the other interfaces. It's an SRP for layers. Layers? Yes. All these service classes, which hide all application logic inside and provide convenient methods to Web, API and other parts, form some structure, which has a lot of names:

- **Service layer**, because of **service** classes.
- **Application layer**, because it contains whole application logic, leaving other jobs such as web request processing to other parts.
- GRASP patterns call this a **Controllers layer**, assuming service classes as a controller.
- there might be some other names.

I like the "controllers layer" name, but it makes sense if this layer only makes a control. For our current state, though, this layer makes almost everything. Service layer also sounds good, but in some projects there are no service classes (see next section). Application layer term is also used in the telecommunications industry and can lead to some misunderstandings. For this book I chose **Application layer**. Because I can.

Service classes or command classes

When an entity has a rich logic and a lot of actions, a service class for this entity can be very big. Every action also wants different dependencies. One action wants email sender and file storage. Another one - some API wrapper. The number of service class constructor parameters grow very quickly and there are a lot of dependencies used only by one-two actions. Usually it happens very quickly in projects, that's why I usually create a class for each application layer action. As far as I know, there is no standard for how these classes should be named. I saw action classes with a **UseCase** suffix, like **CreatePostUseCase**. Also the **Action** suffix is popular: **CreatePostAction**. I usually use the **Command** suffix: **CreatePostCommand**, **PublishPostCommand**, **DeletePostCommand**, etc.

```
final class PublishPostCommand
{
    public function execute($id)
    {
        //...
    }
}
```

In Command bus pattern, the **Command** suffix is usually used only for DTO's, and the **CommandHandler** suffix is for classes that handle the commands.

```
final class ChangeUserPasswordCommand
{
    //...
}

final class ChangeUserPasswordCommandHandler
{
    public function handle(
        ChangeUserPasswordCommand $command)
    {
        //...
    }
}

// or in case of one class to handle
// many commands for some context(User)

final class UserCommandHandler
{
    public function handleChangePassword(
        ChangeUserPasswordCommand $command)
    {
```

```
        //...
    }
}
```

A little remark about long class names. “ChangeUserPasswordCommandHandler - ugh! Such a long name! I don’t want to type it each time!” It will be fully typed only once - during class creation. Every other time the IDE’s auto-complete functionality will help you. A good, fully understandable name is much more important. You can ask any developer “What should the ChangeUserPasswordCommandHandler class do?” Some of them will say “Handle change user password command”. Others: “Change user’s password”. Each of them more or less understands what happens inside this class. That means developers won’t waste time trying to understand these things. It’s much more important than a few seconds for typing.

I’ll use *Service classes for examples in this book to make them a bit more understandable.

6. Error handling

The C language, whose syntax was a base for a lot of modern languages, has a simple error handling convention. If a function should return something and can't do it for some reason, it returns null. If a function should do something and everything is okay, it returns 0, otherwise -1, or some error code. A lot of PHP developers like this simplicity too! Functions which should return a value usually return null if they can't. Action functions usually return boolean values. True for success and false for failure. The code looks like this:

```
final class ChangeUserPasswordDto
{
    private $userId;
    private $oldPassword;
    private $newPassword;

    public function __construct(
        int $userId, string $oldPassword, string $newPassword)
    {
        $this->userId = $userId;
        $this->oldPassword = $oldPassword;
        $this->newPassword = $newPassword;
    }

    public function getUserId(): int
    {
        return $this->userId;
    }

    public function getOldPassword(): string
    {
        return $this->oldPassword;
    }

    public function getNewPassword(): string
    {
        return $this->newPassword;
    }
}

final class UserService
{
    public function changePassword(
        ChangeUserPasswordDto $command): bool
    {
        $user = User::find($command->getUserId());
        if($user === null) {
            return false; // user not found
```

```
        }

        if(!password_verify($command->getOldPassword(),
                            $user->password)) {
            return false; // old password is not valid
        }

        $user->password = password_hash($command->getNewPassword());
        return $user->save();
    }
}

final class UserController
{
    public function changePassword(UserService $service,
                        ChangeUserPasswordRequest $request)
    {
        if($service->changePassword($request->getDto())) {
            // return success web response
        } else {
            // return failure web response
        }
    }
}
```

Well, at least it works. But what if a user wants to know why his response was a failure? There are comments with needed messages but they are useless in runtime. Seems we need something more informative than a boolean value. Let’s try to implement something like this:

```
final class FunctionResult
{
    /** @var bool */
    public $success;

    /** @var mixed */
    public $returnValue;

    /** @var string */
    public $errorMessage;

    private function __construct() {}

    public static function success(
        $returnValue = null): FunctionResult
    {
        $result = new self();
        $result->success = true;
        $result->returnValue = $returnValue;

        return $result;
    }

    public static function error(
        string $errorMessage): FunctionResult
    {
        $result = new self();
        $result->success = false;
```

```
        $result->errorMessage = $errorMessage;

        return $result;
    }
}
```

I created a class for function results. The constructor of this class is private, so its objects can be created only by static factory methods **FunctionResult::success** and **FunctionResult::error**. It's a simple trick called "named" constructors.

```
return FunctionResult::error("Something is wrong");
```

Looks much more natural and more informative than

```
return new FunctionResult(false, null, "Something is wrong");
```

Our code will look like this:

```
class UserService
{
    public function changePassword(
        ChangeUserPasswordDto $command): FunctionResult
    {
        $user = User::find($command->getUserId());
        if($user === null) {
            return FunctionResult::error("User was not found");
        }

        if(!password_verify($command->getOldPassword(),
                            $user->password)) {
            return FunctionResult::error("Old password isn't valid");
        }

        $user->password = password_hash($command->getNewPassword());

        $databaseSaveResult = $user->save();

        if(!$databaseSaveResult->success) {
            return FunctionResult::error("Database error");
        }

        return FunctionResult::success();
    }
}

final class UserController
{
    public function changePassword(UserService $service,
                      ChangeUserPasswordRequest $request)
    {
        $result = $service->changePassword($request->getDto());

        if($result->success) {
```

```
            // return success web response
        } else {
            // return failure web response
            // with $result->errorMessage text
        }
    }
}
```

So, almost every function(even Eloquent's save() method in my imaginary world) returns a **FunctionResult** object with a detailed result of how this function executed. When I was showing this example in one of my seminars, one listener said: "It looks so ugly! Maybe it's better to use exceptions?". I think you agree with him. Just let me show a Go language example:

```
f, err := os.Open("filename.ext")
if err != nil {
    log.Fatal(err)
}
// do something with the open *File f
```

It handles errors almost the same way! There are no exceptions. At least this error handling without exceptions works, but if we want to continue using a **FunctionResult** class in a PHP app, we have to implement call stack and correct logging in each error if-branch. The whole app will be full of **if($result->success)** checks. Definitely not the code I like. I like clean code, code which only describes what should be done, without checking correctness after each step. Actually, the 'validate' method call is clean, because it throws an exception if data is invalid. So, let's start to use them too.

Exceptions

When the user asks our app to do some action, like register a user or cancel an order, the app can execute it successfully or not. If not, there are a ton of reasons why it could go wrong. One of the best illustrations of them is HTTP status codes. There are 2xx and 3xx codes for success request processing, like 200 okay or 302 Found. 4xx and 5xx are used for failed requests, but they are different!

- 4xx are the client errors: 400 Bad Request, 401 Unauthorized, 403 Forbidden, etc.

- 5xx are the server errors: 500 Internal Server Error, 503 Service Unavailable, etc.

So, all failed validation and authorization, not found entities, and trying to change password without knowing the old one are client errors. An API unavailable, file storage error, or database connection issues are the server errors.

There are two main ways how exceptions can be used:

1. Exceptions should be thrown only on the server errors. In cases like “old password is not valid”, the function should return something like **FunctionResult** object.
2. Exceptions should be thrown on both server and client errors.

The first way looks more natural, but passing errors to high levels(from some internal functions to callers, from application layer to controllers) is not very convenient. The second way has unified error handling and cleaner code. There is only one main flow when request processing goes well: user should be fetched from database, the password should match to requests old_password, then password should be changed to requests ‘new_password’ and user entity successfully saved to database. Every step out of this way throws an exception. Validation failed - exception, authorization failed - exception, business logic failure - exception. But later we have to separate client and server errors for correct response generation and logging.

It’s really hard to say which way is better. When the app only extracted the application layer, the second way with exceptions is better. Code is much cleaner this way. Every needed piece of information can be passed through the call stack. But when the application grows and the Domain layer will also be extracted from the Application layer, the exceptions can start to cause troubles. Some exceptions can be thrown and if they won’t be caught in the needed level, they can be interpreted incorrectly in a higher level. So, every internal call should be very accurate and usually surrounded by a try catch structure.

Laravel throws an exception for 404 response, non-authorized(403) error, validation check failed case, so I also chose the second way for this book. We will throw an exception for every case when a requested action cannot be executed.

Let’s try to write code with exceptions:

```
class UserService
{
    public function changePassword(
        ChangeUserPasswordDto $command): void
    {
        $user = User::findOrFail($command->getUserId());

        if(!password_verify($command->getOldPassword(),
                            $user->password)) {
            throw new \Exception("Old password is not valid");
        }

        $user->password = password_hash($command->getNewPassword());

        $user->saveOrFail();
    }
}

final class UserController
{
    public function changePassword(UserService $service,
                        ChangeUserPasswordRequest $request)
    {
        try {
            $service->changePassword($request->getDto());
        } catch(\Throwable $e) {
            // log error
            // return failure web response with $e->getMessage();
        }

        // return success web response
    }
}
```

As you see, even in this super simple example, **UserService::changePassword** method code looks much cleaner. All cases outside of main flow just throw an exception, which stops following execution and goes to a high level(controller). Eloquent also has methods for this style of code: **findOrFail()**, **firstOrFail()** and some other ***OrFail()** methods. But this code has several problems:

1. **Exception::getMessage()** is not a good message to show to a user. “Old password is not valid” message is okay, but “Server Has Gone

Away (error 2006)” is definitely not.

2. Each server exception should be logged. Small applications use log files. When the application becomes more popular, exceptions can be thrown each second. Some exceptions signal a problem in the code and have to be fixed immediately. Some exceptions are okay. Internet not ideal, API’s regularly answer with error HTTP codes or just timeouts. Developers have to react only if the rate of exceptions with some APIs becomes too frequent.
   In that case, it is better to use special services to store the logs. These services allows developers to group and work with exceptions more conveniently. Just ask Google “error monitoring services” and it helps to find some of them. Big companies build special solutions for storing and analyzing logs from all of their servers. Some companies prefer to store client exceptions too, but storing, for example, 404 errors looks strange for me (it may be stored in http server log, but not in application logs). Anyway, we have to separate server and client exceptions and process them differently.

Base exception class

The first step is to create an exception class for all business-related exceptions (like “Old password is not valid”). PHP has a **DomainException** class that can be used for it, but other code also could use it and it might become a big surprise. Especially if we won’t log them. So, better to create our own class. Let say **BusinessException**.

```
class BusinessException extends \Exception
{
    /**
    * @var string
    */
    private $userMessage;

    public function __construct(string $userMessage)
    {
        $this->userMessage = $userMessage;
        parent::__construct("Business exception");
    }

    public function getUserMessage(): string
    {
        return $this->userMessage;
    }
}
```

```
// Now password verification failed case will throw new exception

if(!password_verify($command->getOldPassword(), $user->password)) {
    throw new BusinessException("Old password is not valid");
}

final class UserController
{
    public function changePassword(UserService $service,
                        ChangeUserPasswordRequest $request)
    {
        try {
            $service->changePassword($request->getDto());
        } catch(BusinessException $e) {
            // return failure web response
            // (with 400 HTTP code)
            // with $e->getUserMessage();
        } catch(\Throwable $e) {
            // log error

            // return failure web response(with 500 HTTP code)
            // with "Houston, we have a problem";
            // Not with real exception message
        }

        // return success web response
    }
}
```

This code catches **BusinessException** and shows the error message to the user. For all other exceptions some “Inner exception” message will be shown to user and the exception will be logged. This code works correctly. User (or API caller) sees correct errors. Each error which should be logged will be logged. But this “catch” part will be repeated in each controller action. For all web actions it will be the same. For all API actions, too. We can extract it to higher level.

Global handler

Laravel(and almost all other frameworks) has a global exceptions handler and it is the best place to extract common “catch” logic. Laravel’s **app/Exceptions/Handler.php** class has 2 close responsibilities: reporting and rendering exceptions.

```
namespace App\Exceptions;

class Handler extends ExceptionHandler
{
    protected $dontReport = [
        // This means BusinessException
        // should not be reported,
        // but it will be rendered
```

```
        BusinessException::class,
    ];

    public function report(Exception $e)
    {
        if ($this->shouldReport($e))
        {
            // Here is the best place to
            // integrate external exception
            // monitoring services
        }

        // This will log exception to log file, by default
        parent::report($e);
    }

    public function render($request, Exception $e)
    {
        if ($e instanceof BusinessException)
        {
            if($request->ajax())
            {
                $json = [
                    'success' => false,
                    'error' => $e->getUserMessage(),
                ];

                return response()->json($json, 400);
            }
            else
            {
                return redirect()->back()
                       ->withInput()
                       ->withErrors([
                           'error' => trans($e->getUserMessage())]);
            }
        }

        // Default rendering,
        // like showing 404 page for 404 error,
        // "Oops" page for 500 error, etc.
        return parent::render($request, $e);
    }
}
```

Simple example of Handler class. ‘report’ method can be used for some additional reporting. “catch” part from controller actions was moved to ‘render’ method. Here all BusinessException objects will be filtered and correct responses will be generated both for web and API parts. For CLI custom errors rendering undocumented **renderForConsole**(**$output**, **Exception $e**) method might be overwritten in **Handler** class. Now controller action code becomes much prettier:

```
final class UserController
{
    public function changePassword(UserService $service,
```

```
                        ChangeUserPasswordRequest $request)
    {
        $service->changePassword($request->getDto());

        // return success web response
    }
}
```

Checked and unchecked exceptions

Let’s take a look at the **UserService::changePassword** method. What kind of exceptions can be thrown there?

- **IlluminateDatabaseEloquentModelNotFoundException** if there is no user with this id
- **IlluminateDatabaseQueryException** if database query couldn’t be executed
- **AppExceptionsBusinessException** if old password is wrong
- **TypeError** if somewhere deep in the code function **foo**(**SomeClass $x**) will get **$x** parameter value with another type
- **Error** if some **$var->method()** will be called when $var is null
- a lot of other possible exceptions

From the **UserService::changePassword** callee point of view, some of these exceptions, like **Error**, **TypeError**, **QueryException** are definitely out of context. Http controller doesn’t know what to do with them. The only possible reaction - to show some message, like “Something happened and I don’t know what to do!” But some of them make sense for this controller. BusinessException means that something was wrong with the logic inside and it has a special message for the user. Controller definitely knows what to do if this exception was thrown. The same can be said about **ModelNotFoundException**. Controller can show 404 error in that case. So, 2 types of errors:

1. Errors which can be effectively processed by caller
2. Other errors

It will be good if the first set of errors will be caught right in the caller, because in this place it has maximum information on how to handle this exception. If it will be caught in higher layers it will be hard to react correctly. Let’s keep this in mind and take a look at Java.

```
public class Foo
{
    public void bar()
    {
        throw new Exception("test");
    }
}
```

This code won’t even be compiled. Compiler’s message: “Error:(5, 9) java: unreported exception java.lang.Exception; must be caught or declared to be thrown” There are 2 ways to fix that. Catch it:

```
public class Foo
{
    public void bar()
    {
        try {
            throw new Exception("test");
        } catch(Exception e) {
            // do something
        }
    }
}
```

Or declare this exception in method signature:

```
public class Foo
{
    public void bar() throws Exception
    {
        throw new Exception("test");
    }
}
```

In that case, every method caller has to catch this exception or, again, describe it in its own signature:

```
public class FooCaller
{
    public void caller() throws Exception
    {
        (new Foo)->bar();
    }

    public void caller2()
    {
        try {
            (new Foo)->bar();
        } catch(Exception e) {
            // do something
        }
    }
}
```

Of course, working like that with ALL exceptions can be very annoying. Java has **checked** exceptions, which have to be declared or caught, and **unchecked** exceptions, which can be thrown without any conditions. Take a look to the Java main exception classes tree (PHP starting from 7 version has the same structure):

```
          Throwable(checked)
         /          \
Error(unchecked)  Exception(checked)
                        \
                     RuntimeException(unchecked)
```

All Throwable and Exception derived classes become checked exceptions. Except Error and RuntimeException and all their derived classes.

```
public class File
{
    public String getCanonicalPath() throws IOException {
        //...
    }
}
```

What does the **getCanonicalPath** method signature say to the developer? It doesn't have any parameters. Returns **String** object. Throws checked exception **IOException** and maybe some unchecked exceptions.

Returning to our 2 error types:

1. Errors which can be effectively processed by caller
2. Other errors

Checked exceptions are created for the first error type. Unchecked for second. Caller has to do something with a checked exception and this strictness helps to make the code which processes exceptions as correct as possible.

Well, Java has this feature. PHP doesn't. Why am I still talking about it? The IDE I use, PHPStorm, imitates this Java behaviour.

```
class Foo
{
    public function bar()
    {
        throw new Exception();
```

```
    }
}
```

PHPStorm will highlight ‘throw new Exception();’ with warning: ‘Unhandled Exception’. The same two ways to remove this warning:

1. Catch the exception
2. Describe it in @throws tag on methods phpDoc:

```
class Foo
{
    /**
     * @throws Exception
     */
    public function bar()
    {
        throw new Exception();
    }
}
```

The unchecked exception classes list is configurable. By default there are: **Error**, **RuntimeException** and **LogicException** classes. Throwing them and their derived classes doesn’t create any warnings.

With all this information we now can decide how to build our exception classes. I definitely want to inform the **UserService::changePassword** caller about:

1. **ModelNotFoundException**, when needed user wasn’t found
2. **BusinessException**, this exception sends the error message to the user and must be processed correctly. All other exceptions can be processed later.
   So, in an ideal world:

```
class ModelNotFoundException extends \Exception
{...}

class BusinessException extends \Exception
{...}

final class UserService
{
    /**
     * @param ChangeUserPasswordDto $command
     * @throws ModelNotFoundException
     * @throws BusinessException
     */
    public function changePassword(
```

```
        ChangeUserPasswordDto $command): void
    {...}
}
```

But we already moved all exceptions handling to the Handler class and now we have to copy **@throws** tags in controller actions:

```
final class UserController
{
    /**
     * @param $userId
     * @param UserService $service
     * @param Request $request
     * @throws ModelNotFoundException
     * @throws BusinessException
     */
    public function changePassword(UserService $service,
                        ChangeUserPasswordRequest $request)
    {
        $service->changePassword($request->getDto());

        // return success web response
    }
}
```

Not very convenient. Even if PHPStorm can auto-create all these phpDocs. Returning to our non-ideal world: Laravel's **ModelNotFoundException** already inherited from **RuntimeException**. So, by default it is unchecked. It's reasonable, because Laravel already has this exception processing deep in frameworks Handler class. In our current state, it may be better also to do the same trade-off:

```
class BusinessException extends \RuntimeException
{...}
```

and forget about **@throws** tag assuming all **BusinessException** exceptions will be processed in the global **Handler** class.

Actually, this is one of the reasons why modern languages don't have a checked exceptions feature and most Java developers don't use them. Another reason: some libraries just make "throws Exception" in their methods. "throws Exception" statement doesn't give any useful information. It just forces client code to catch exceptions or repeat this useless "throws Exception" in its own signature.

I’ll return to exceptions in the Domain layer chapter, when this way with unchecked exceptions will become not very convenient.

Conclusion

Function or method returning more than one possible type, nullable or boolean result(ok/not ok) makes a callee code more dirty. Each callee has to check the result(‘!== null’ or ‘if($result)’) right after call. Code with exceptions looks cleaner:

```
// Without exception
$user = User::find($command->getUserId());
if($user === null) {
    // process the error
}

$user->doSomething();


// With exception
$user = User::findOrFail($command->getUserId());
$user->doSomething();
```

On the other hand, using something like **FunctionResult** objects gives much more control to the developer. For example, careless **findOrFail** will show 404 error, instead of the correct error message. Exceptions should be used very accurately.

7. Validation

> "…But, now you come to me, and you say: "Don Corleone, give me justice." But you don't ask with respect. You don't offer friendship…"

Database related validation

As usual, I'm starting a new chapter with the code example containing all of the practices described in the book up to this point. Create post use case:

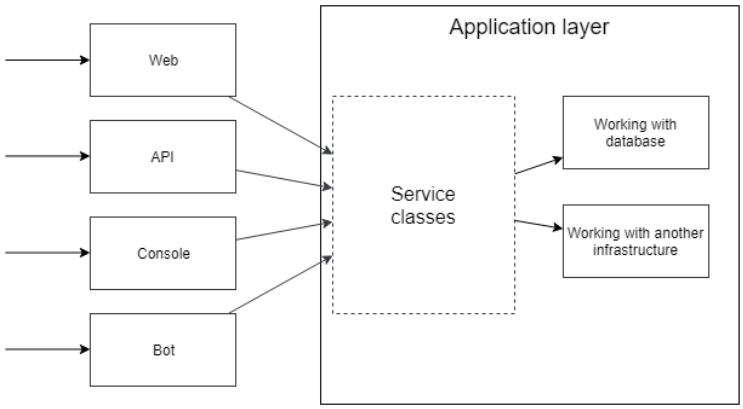
```
class PostController extends Controller
{
    public function create(Request $request, PostService $service)
    {
        $this->validate($request, [
            'category_id' => 'required|exists:categories',
            'title' => 'required',
            'body' => 'required',
        ]);

        $service->create(/* DTO */);

        //...
    }
}

class PostService
{
    public function create(CreatePostDto $dto)
    {
        $post = new Post();
        $post->category_id = $dto->getCategoryId();
        $post->title = $dto->getTitle();
        $post->body = $dto->getBody();

        $post->saveOrFail();
    }
}
```

Later, a "soft delete" feature will be added to the **Category** model. The soft delete pattern suggests to mark rows as deleted by special field instead of actually deleting by DELETE SQL command. In Laravel, it can be implemented simply by adding the **SoftDeletes** trait to the **Category** class.

Other parts of application did not change and it works as usual. But our validation rule for the category_id field is broken! It accepts deleted categories. We have to change it (and all other ‘category_id’ field validations in the entire app).

```
$this->validate($request, [
    'category_id' => [
        'required|exists:categories,id,deleted_at,null',
    ],
    'title' => 'required',
    'body' => 'required',
]);
```

New requirement: add ‘archived’ field to Category model and don’t allow post creation in archived categories. How can this be implemented? Changing the validation rules again? Add $query->where(‘archived’, 0); there? But the HTTP part of the application haven’t changed! We are just changing our business logic (with archived field) and database storing (with soft delete). Why do all of these changes affect HTTP requests? It’s another example of high coupling. As you remember, we moved all work with database to the application layer, but validation rules were forgotten.

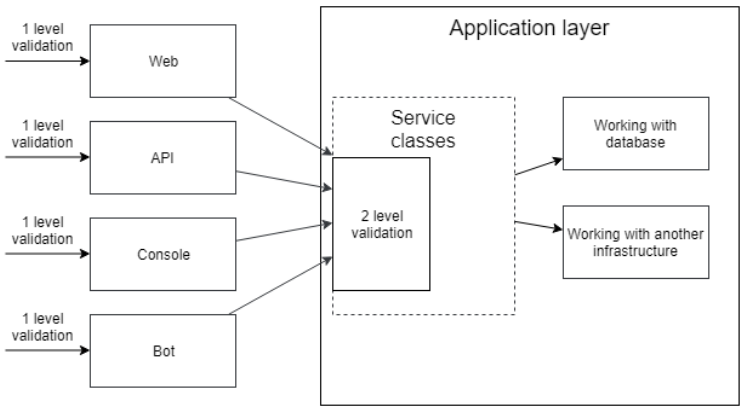


Let’s separate validations:

```
$this->validate($request, [
    'category_id' => 'required',
    'title' => 'required',
    'body' => 'required',
]);

class PostService
```

```
{
    public function create(CreatePostDto $dto)
    {
        $category = Category::find($dto->getCategoryId());

        if($category === null) {
            // throw "Category not found" exception
        }

        if($category->archived) {
            // throw "Category archived" exception
        }

        $post = new Post();
        $post->category_id = $category->id;
        $post->title = $dto->getTitle();
        $post->body = $dto->getBody();

        $post->saveOrFail();
    }
}
```

Okay, now validation is in its correct place. **PostService::create** doesn't trust such important validation to high-level callers and validates by itself. As a bonus, the application now has more informative error messages.



Two levels of validation

In the previous example, we have some validation both in the request class and in the application layer. I also said that **PostService::create** doesn't trust validation to callers, but it still trusts. It makes:

```
$post->title = $request->getTitle();
```

and here we can’t be 100% sure that **getTitle()** will return a non-empty string! Yes, it will be checked with ‘required’ validation, but it’s so far away in another layer of our app. **PostService::create** can be called from another place and this validation can be forgotten there. Let’s use a user registration use case as a better example of this:

```
class RegisterUserDto
{
    /** @var string */
    private $name;

    /** @var string */
    private $email;

    /** @var DateTime */
    private $birthDate;

    public function __construct(
        string $name, string $email, DateTime $birthDate)
    {
        $this->name = $name;
        $this->email = $email;
        $this->birthDate = $birthDate;
    }

    public function getName(): string
    {
        return $this->name;
    }

    public function getEmail(): string
    {
        return $this->email;
    }

    public function getBirthDate(): DateTime
    {
        return $this->birthDate;
    }
}

class UserService
{
    public function register(RegisterUserDto $request)
    {
        $existingUser = User::whereEmail($request->getEmail())
            ->first();

        if($existingUser !== null) {
            throw new UserWithThisEmailAlreadyExists(...);
        }

        $user = new User();
        $user->name = $request->getName();
        $user->email = $request->getEmail();
        $user->birthDate = $request->getBirthDate();

        $user->saveOrFail();
```

```
    }
}
```

By moving to DTO, we had to forget about the web request validation. Yes, there is a validation with ‘required’ and ‘email’ fields in the web interface, but another interface, like bot, can provide corrupted data.

```
$userService->register(new RegisterUserDto('', '', new DateTime()));
```

CreateUserCommand can’t be sure that **getName()** will return a non-empty string and **getEmail()** will return a string with a correct email. What to do? Some duplicated validation can be used in command:

```
class UserService
{
    public function register(RegisterUserDto $request)
    {
        if(empty($request->getName())) {
            throw //
        }

        if(!filter_var($request->getEmail(),
                       FILTER_VALIDATE_EMAIL)) {
            throw //
        }

        //...
    }
}
```

The same validation can be used in DTO’s constructor, but there will be a lot of duplicated validation. There will be a lot of places in the application with name and email values. I can suggest two ways to solve this problem.

Validation by annotations

Symfony framework has a great component for validation: **symfony/validator**. To use it not in symfony (today in 2019), you have to install **symfony/validator**, **doctrine/annotations** and **doctrine/cache** composer packages and make some initialization for the Annotation loader. Rewriting our **RegisterUserDto**:

```
use Symfony\Component\Validator\Constraints as Assert;

class RegisterUserDto
{
    /**
     * @Assert\NotBlank()
```

```
     * @var string
     */
    private $name;

    /**
     * @Assert\NotBlank()
     * @Assert\Email()
     * @var string
     */
    private $email;

    /**
     * @Assert\NotNull()
     * @var DateTime
     */
    private $birthDate;

    // Constructor and getters are the same
}
```

Then, let's bind **ValidatorInterface** instance:

```
$container->bind(
    \Symfony\Component\Validator\Validator\ValidatorInterface::class,
    function() {
        return \Symfony\Component\Validator\Validation
            ::createValidatorBuilder()
                ->enableAnnotationMapping()
                ->getValidator();
});
```

Now, we can use it in the service class:

```
class UserService
{
    /** @var ValidatorInterface */
    private $validator;

    public function __construct(ValidatorInterface $validator)
    {
        $this->validator = $validator;
    }

    public function register(RegisterUserDto $dto)
    {
        $violations = $this->validator->validate($dto);

        if (count($violations) > 0) {
            throw new ValidationException($violations);
        }

        $existingUser = User::whereEmail($dto->getEmail())->first();

        if($existingUser !== null) {
            throw new UserWithThisEmailAlreadyExists(...);
        }

        $user = new User();
```

```
        $user->name = $dto->getName();
        $user->email = $dto->getEmail();
        $user->birthDate = $dto->getBirthDate();

        $user->saveOrFail();
    }
}
```

ValidatorInterface::validate method returns the list of violations. If it's empty - everything is okay. Otherwise, the **ValidationException** will be thrown. Using this explicit validation, the application layer action can be sure about data validity. Also, as a big advantage, the Web and API layers can remove their own validation. Anyway, all request data will be validated by the Application layer. Looks fantastic, but there are some problems.

Http request data != Application layer DTO data

First, web request data does not always repeat application layer DTO. When Web UI asks user to change password, it shows a field for old password, a field for new password and a field to repeat new password. Web request validation should check that "new password" and "repeat new password" fields values are the same. Application layer doesn't care about it. It only gets old password and new password values and does the job. Another case: one field for some request, email for example, will be filled by current user's email. If this email is somehow empty, application layer will return validation exception like "Wrong email!" and it will be shown to the user. User will see "Wrong email" error in the form that doesn't even have this field! It's definitely not an example of good UI practices.

Complex data structures

Imagine some **CreateTaxiOrderDto**. It's an avia taxi, so orders can be from one country to another. **fromHouse**, **fromStreet**, **fromCity**, **fromState**, **fromCountry**, **toHouse**, **toStreet**, **toCity**,… Huge DTO with a lot of fields duplicating each other, depending on each other. House fields values mean nothing without street fields values. Street without city, state and country. And imagine the chaos when the taxi becomes inter-galactic!

Validation of some fields can be dependent on others. User leaves his contact information, email and/or phone. The business requirement is to have at least one of them: email or phone. So, in each DTO we have to

implement something like ‘required_without’ Laravel rule for both of these fields. Not very convenient.

Value objects

There is a solution right in the **RegisterUserDto**. We don’t store **$birthDay**, **$birthMonth** and **$birthYear** there. We just store a **DateTime** object! We don’t validate it each time. It always stores a correct date and time value. When we compare two **DateTime** values, we don’t compare their years, months, etc. It is better to use **diff()** method or comparison operators. So, all knowledge about date time value type is consolidated in one class. Let’s try to do the same with **RegisterUserDto** values:

```
final class Email
{
    /** @var string */
    private $email;

    private function __construct(string $email)
    {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException(
                'Email ' . $email . ' is not valid');
        }

        $this->email = $email;
    }

    public static function create(string $email)
    {
        return new static($email);
    }

    public function value(): string
    {
        return $this->email;
    }
}

final class UserName
{
    /** @var string */
    private $name;

    private function __construct(string $name)
    {
        if (/* Some validation of $name value*.
            It depends on project requirements. */) {
            throw new InvalidArgumentException(
                'Invalid user name: ' . $name);
        }

        $this->name = $name;
    }
```

```
    public static function create(string $name)
    {
        return new static($name);
    }

    public function value(): string
    {
        return $this->name;
    }
}

final class RegisterUserDto
{
    // fields and constructor

    public function getUserName(): UserName
    {...}

    public function getEmail(): Email
    {...}

    public function getBirthDate(): DateTime
    {...}
}
```

Yes, creating a class for each input value is not what developers are dreaming about. But it's just a natural way of application decomposition. Instead of validating strings and using strings as values, which can result in unvalidated data somewhere, these classes only allow you to have correct, validated data. This pattern is called **Value Object**(VO). Now everyone can trust in values returned by these getters. **getEmail()** doesn't return a meaningless string value. It returns a real email value, which can be used without any fear. **UserService** class now can trust the values and use them:

```
final class UserService
{
    public function register(RegisterUserDto $dto)
    {
        //...
        $user = new User();
        $user->name = $dto->getName()->value();
        $user->email = $dto->getEmail()->value();
        $user->birthDate = $dto->getBirthDate();

        $user->saveOrFail();
    }
}
```

Yes, **->value()** calls look a bit weird. I think it can be solved by overriding **__toString()** magic method of **Email** and **UserName** class, but I'm not be sure if it works with Eloquent values. Even if it works, it's implicit magic. I

don’t like these kinds of solutions. Later we will try to deal with this problem.

VO as composition of values

Email and **UserName** value objects are basically just wrappers for strings. Value objects are a much wider concept. Geo-point is a structure with two float values: Latitude and Longitude.
Usually nobody is interested in a Latitude value without a Longitude(if the first one is not a 90 or -90 :)). By creating a **GeoPoint**(float **$latitude**, float **$longitude**) value object, almost all program code can work with Geo coordinates as one type, without remembering that it’s two doubles. The same thing we do with DateTime.

```
final class GeoPoint
{
    /** @var float */
    private $latitude;

    /** @var float */
    private $longitude;

    public function __construct(float $latitude, float $longitude)
    {
        $this->latitude = $latitude;
        $this->longitude = $longitude;
    }

    public function getLatitude(): float
    {
        return $this->latitude;
    }

    public function getLongitude(): float
    {
        return $this->longitude;
    }

    public function isEqual(GeoPoint $other): bool
    {
        // Just for example
        return $this->getDistance($other)->getMeters() < 10;
    }

    public function getDistance(GeoPoint $other): Distance
    {
        // Calculating distance between $this and $other points
    }
}

final class City
{
    //...
```

```
    public function setCenterPoint(GeoPoint $centerPoint)
    {
        $this->centerLatitude = $centerPoint->getLatitude();
        $this->centerLongitude = $centerPoint->getLongitude();
    }

    public function getCenterPoint(): GeoPoint
    {
        return new GeoPoint(
            $this->centerLatitude, $this->centerLongitude);
    }
}
```

In this example, City class encapsulated latitude and longitude storing details and provides only a GeoPoint instance. All other app code can use it to calculate distances between points and other things without thinking about how this class works.

Other examples of value objects:

- **Money**(int **amount**, Currency **currency**)
- **Address**(string **street**, string **city**, string **state**, string **country**, string **zipcode**)

Have you noticed that in the last example I'm trying to not use primitive types, like string, int, float? **getDistance()** returns a **Distance** object, not int or float. **Distance** object has methods like **getMeters()**: float or **getMiles()**: float. **Distance::isEqual**(**Distance $other**) can be used to compare two distances. It's a Value Object, too! Well, sometimes it's over-engineering. For some projects, **Point::getDistance()**: float returning a value in meters will be enough. I just wanted to show an example of which I call 'Thinking by objects'. We will return to Value Objects later in this book. As you maybe understand, VO is too powerful a thing to use only as DTO fields.

Value object is not for user data validation

Current DTO and controller code:

```
final class RegisterUserDto
{
    // fields and constructor

    public function getUserName(): UserName
    {...}

    public function getEmail(): Email
```

```
    {...}

    public function getBirthDate(): DateTime
    {...}
}

final class UserController extends Controller
{
    public function register(
        Request $request, UserService $service)
    {
        $this->validate($request, [
            'name' => 'required',
            'email' => 'required|email',
            'birth_date' => 'required|date',
        ]);

        $service->register(new RegisterUserDto(
            UserName::create($request['name']),
            Email::create($request['email']),
            DateTime::createFromFormat('some format', $request)
        ));

        //return ok response
    }
}
```

It’s easy to find duplications in this code. **Email** value has first been validated with Laravel validation **$this->validate()** and then in the **Email** constructor:

```
final class Email
{
    private function __construct(string $email)
    {
        if (!filter_var($email, FILTER_VALIDATE_EMAIL)) {
            throw new InvalidArgumentException(
                'Email ' . $email . ' is not valid');
        }

        $this->email = $email;
    }

    public static function create(string $email)
    {
        return new static($email);
    }
}
```

The idea of removing this duplication looks interesting. **$this->validate()** call can be removed and replaced by catching **InvalidArgumentException** in the global error handler. This idea looks good only at first sight. As I mentioned before, “Http request data != Application layer DTO data”. User can get validation error messages not about his data in this case.

Value objects can be used not only for validation purposes. There are cases when an error inside the application will be interpreted as a validation error, and this is not the experience the user wants to have. If you remember, PhpStorm by default has 3 root classes for unchecked exceptions: **Error**, **RuntimeException** and **LogicException**:

- **Error** represents some PHP language inner exceptions, like **TypeError**, **ParseError**, etc.
- **RuntimeException** represents an error which can be thrown only in runtime and the reason is not in our code(like database connection issues).
- **InvalidArgumentException** extends **LogicException**. **LogicException** description from php documentation: “Exception that represents error in the program logic. This kind of exception should lead directly to a fix in your code.” So, if code is written well, it should never throw **LogicException**. That means the checks in VO constructors are just to be sure that the data was already checked earlier. It’s kind of an application code verification, not user input validation.

Conclusion

Moving business logic to the application layer results in some issues with data validation. Web **FormRequest** objects can’t be used anymore, and some kind of Data transfer objects should be used instead (it might be the usual PHP arrays or special DTO classes). If the application layer DTO always represents user input, then the user input validation can be moved to the application layer and implemented by **symfony/validator** or another validation package. It will be a bit dangerous and sometimes not convenient with complex data.

Validation can be left in Web, API and other parts. DTO will have data without any additional checks. So, the application layer should just trust the data, processed by callers. From my experience, it works only in small projects. Big projects written by a team of developers and this approach will always lead to some incidents with corrupted data in the database or just runtime errors.

The Value Object pattern requires some additional coding and “thinking by objects” from developers, but they provide the most safe and natural data representation. As always, it’s a trade-off between short term and long term productivity.

8. Events

> Procrastinator's rule: if something can be postponed, it should be postponed.

Application layer action often contains the main action and some secondary actions. User registration contains user entity creation and register email sending actions. Post text updating contains updating **$post->text** with saving and, for example, **Cache::forget** calls for deleting old values from cache. Pseudo-real example: site with polls. Poll creation:

```
final class PollService
{
    /** @var BadWordsFilter */
    private $badWordsFilter;

    public function __construct(BadWordsFilter $badWordsFilter
        /*, ... other dependencies */)
    {
        $this->badWordsFilter = $badWordsFilter;
    }

    public function create(PollCreateDto $request)
    {
        $poll = new Poll();
        $poll->question = $this->badWordsFilter->filter(
            $request->getQuestion());
        //...
        $poll->save();

        foreach($request->getOptionTexts() as $optionText)
        {
            $pollOption = new PollOption();
            $pollOption->poll_id = $poll->id;
            $pollOption->text =
                    $this->badWordsFilter->filter($optionText);
            $pollOption->save();
        }

        // Call sitemap generator

        // Notify external API about new poll
    }
}
```

Here is a poll and options creation action with filtering for bad words in all texts and some post-actions. Poll object is not simple. It's absolutely useless if it has no options. We have to take care about it's consistency. This little period of time when **Poll** object already created and saved to the database before **PollOption** objects were also saved to the database and added to **Poll** object is very dangerous.

Database transactions

First problem - database. Some error can happen and the **Poll** object will be saved but the **PollOption** objects not. Or at least not all of them. All modern database engines created for storing data whose consistency is important have transaction support. Database transactions guarantee the consistency in the database. We can run some queries under them and all of them will be executed. If some error happens (database error or exception in user code) all queries in the transaction will be rolled back and have no effect on the database data. Looks like a solution:

```
final class PollService
{
    /** @var DatabaseConnection */
    private $connection;

    public function __construct(..., DatabaseConnection $connection)
    {
        ...
        $this->connection = $connection;
    }

    public function create(PollCreateDto $request)
    {
        $this->connection->transaction(function() use ($request) {
            $poll = new Poll();
            $poll->question = $this->badWordsFilter->filter(
                    $request->getQuestion());
            //...
            $poll->save();

            foreach($request->getOptionTexts() as $optionText) {
                $pollOption = new PollOption();
                $pollOption->poll_id = $poll->id;
                $pollOption->text =
                    $this->badWordsFilter->filter($optionText);
                $pollOption->save();
            }

            // Call sitemap generator

            // Notify external API about new poll
        });
```

```
    }
}
```

Okay, now our data will be consistent in the database, but this transaction magic is not free for database engines. When we run queries in transaction, DBMS have to store two versions of data: for successful and fail cases. In high-load projects there may be hundreds of concurrent transactions and when the execution time of each transaction is long, it may drastically reduce performance. For a non-high-load project it's not very important, but it is still better to make it a habit to run transactions as fast as possible. All post actions should definitely be moved outside of the transaction. Bad words filter can be our inner service or some external API call (which can take a lot of time). It also should be moved outside of the transaction.

```
final class PollService
{
public function create(PollCreateDto $request)
{
    $filteredRequest = $this->filterRequest($request);

    $this->connection->transaction(
        function() use ($filteredRequest) {
            $poll = new Poll();
            $poll->question = $filteredRequest->getQuestion();
            //...
            $poll->save();

            foreach($filteredRequest->getOptionTexts()
                                as $optionText) {
                $pollOption = new PollOption();
                $pollOption->poll_id = $poll->id;
                $pollOption->text = $optionText;
                $pollOption->save();
            }
        });

    // Call sitemap generator

    // Notify external API about new poll
}

private function filterRequest(
    PollCreateDto $request): PollCreateDto
{
    // filters the request texts
    // and return the same DTO but with filtered data
}
}
```

Queues

Second problem - request execution time. Application should respond as fast as possible. Poll creation contains some heavy operations like sitemap update and external API calling. Common solution - postpone some actions using queues. Instead of executing some heavy action in the web request, the application can create a task to execute this action and put it to a queue. A queue can be a database table, Redis list or special queue solutions, like RabbitMQ. Laravel suggests several ways to work with queues. One of them: jobs. As I said before, usually there is one main action and some secondary actions. The main action here is creating a Poll object and it can't be executed without filtering texts. All post-actions can be executed later. Actually, in some rare cases when it takes too long, the whole application layer action can be executed in a queued job. But this is not our case.

```
final class SitemapGenerationJob implements ShouldQueue
{
    public function handle()
    {
        // Call sitemap generator
    }
}

final class NotifyExternalApiJob implements ShouldQueue {}

use Illuminate\Contracts\Bus\Dispatcher;

final class PollService
{
    /** @var Dispatcher */
    private $dispatcher;

    public function __construct(..., Dispatcher $dispatcher)
    {
        ...
        $this->dispatcher = $dispatcher;
    }

    public function create(PollCreateDto $request)
    {
        $filteredRequest = $this->filterRequest($request);

        $poll = new Poll();
        $this->connection->transaction(...);

        $this->dispatcher->dispatch(
            new SitemapGenerationJob());
        $this->dispatcher->dispatch(
            new NotifyExternalApiJob($poll->id));
    }
}
```

If the Laravel job class implements empty interface **ShouldQueue**, it's execution will be queued. Well, this code now executes pretty fast, but I still don't like it. There can be a lot of post-actions in big projects and the service class starts to know too much. It executes the main action and knows about all the post-actions. From a high-level point of view, poll creation action should not know about sitemap generation. If it hypothetically will be moved to another project, without sitemaps it couldn't work there. Also, in a project with a lot of actions and post-actions, developers need a convenient configuration for which post-actions should be called after which action.

Events

Instead of directly calling all post-actions, the service action can just inform the application that something happened. The application then can react by executing all needed post-actions. Laravel has an events feature for that.

```
final class PollCreated
{
    /** @var int */
    private $pollId;

    public function __construct(int $pollId)
    {
        $this->pollId = $pollId;
    }

    public function getPollId(): int
    {
        return $this->pollId;
    }
}

use Illuminate\Contracts\Events\Dispatcher;

final class PollService
{
    /** @var Dispatcher */
    private $dispatcher;

    public function __construct(..., Dispatcher $dispatcher)
    {
        ...
        $this->dispatcher = $dispatcher;
    }

    public function create(PollCreateDto $request)
    {
        // ...

        $poll = new Poll();
```

```
        $this->connection->transaction(
            function() use ($filteredRequest, $poll) {
                // ...
            });

        $this->dispatcher->dispatch(new PollCreated($poll->id));
    }
}

final class SitemapGenerationListener implements ShouldQueue
{
    public function handle($event)
    {
        // Call sitemap generator
    }
}

final class EventServiceProvider extends ServiceProvider
{
    protected $listen = [
        PollCreated::class => [
            SitemapGenerationListener::class,
            NotifyExternalApiListener::class,
        ],
    ];
}
```

Application layer now only notifies the application that something is happened. **PollCreated** or something else. The application has configuration on how to react to these events. **Listener** class handles the subscribed events. **ShouldQueue** interface works the same as with jobs, as a mark when it should be processed: immediately or queued. Events is a powerful mechanism, but it has a lot of traps.

Using Eloquent events

Laravel fires a lot of system events. Cache events: **CacheHit**, **CacheMissed**, etc. Notification events: **NotificationSent**, **NotificationFailed**, etc. Eloquent also fires its own events. Example from documentation:

```
class User extends Authenticatable
{
    /**
     * The event map for the model.
     *
     * @var array
     */
    protected $dispatchesEvents = [
        'saved' => UserSaved::class,
        'deleted' => UserDeleted::class,
    ];
}
```

UserSaved event will be fired each time the User model will be “saved” to database. “Saved” means any insert or update SQL query done by this entity class. Using these events has a lot of disadvantages.

UserSaved is not a good name for this event. **UsersTableRowInsertedOrUpdated** is more correct. But even this is sometimes wrong. This event won’t be fired in bulk update operations. Deleted event won’t be fired if the row will be deleted by the database cascade delete operation. The main problem is that these events are infrastructure events, database rows events, but they are used as business or domain events. The difference is easy to see in our poll creation example:

```
final class PollService
{
public function create(PollCreateDto $request)
{
    //...
    $this->connection->transaction(function() use (...) {
        $poll = new Poll();
        $poll->question = $filteredRequest->getQuestion();
        //...
        $poll->save();

        foreach($filteredRequest->getOptionTexts() as $optionText){
            $pollOption = new PollOption();
            $pollOption->poll_id = $poll->id;
            $pollOption->text = $optionText;
            $pollOption->save();
        }
    });
    //...
}
}
```

$poll->save(); call will fire ‘saved’ event. First problem here is that the **Poll** object is still not ready and inconsistent. It doesn’t have options yet. If the event listener wants to build, for example, an email about a new poll, it will definitely try to fetch all options. Yes, for queued listeners it’s not a problem, but in developers’ machines the **QUEUE_DRIVER** value is ‘sync’, so all queued jobs/listeners become “not queued” and execute immediately. I strongly recommend to avoid solutions which work “sometimes, in some conditions”.

The second problem that these events will be fired inside a transaction. Running listeners immediately or putting them to queue will make the transaction much longer and more fragile. Even worse, that event, like

PollCreated, will be fired, but the transaction can be rolled back for some reason! Email with poll, which wasn't even created, will be sent to the user. I found Laravel packages that collect these events, wait for the transaction to be committed, and only then run them (google "Laravel transactional events"). So, they are trying to fix both of these problems, but it looks so unnatural! The simple idea to fire normal domain event **PollCreated** after the transaction is successfully committed is much better.

Entities as event fields

The second trap is providing entity objects as an event field.

```php
final class PollCreated
{
    /** @var Poll */
    private $poll;

    public function __construct(Poll $poll)
    {
        $this->poll = $poll;
    }

    public function getPoll(): Poll
    {
        return $this->poll;
    }
}

final class PollService
{
    public function create(PollCreateDto $request)
    {
        // ...
        $poll = new Poll();
        // ...
        $this->dispatcher->dispatch(new PollCreated($poll));
    }
}

final class SendPollCreatedEmailListener implements ShouldQueue
{
    public function handle(PollCreated $event)
    {
        // ...
        foreach($event->getPoll()->options as $option)
        {...}
    }
}
```

It's just an example of listener, which uses some **HasMany** relation values. This code works okay. When the listener will ask **$event->getPoll()-**

>options, Eloquent makes a query to the database and fetches fresh values for this relation. Another example:

```
final class PollOptionAdded
{
    /** @var Poll */
    private $poll;

    public function __construct(Poll $poll)
    {
        $this->poll = $poll;
    }

    public function getPoll(): Poll
    {
        return $this->poll;
    }
}

final class PollService
{
public function addOption(PollAddOptionDto $request)
{
    $poll = Poll::findOrFail($request->getPollId());

    if($poll->options->count() >= Poll::MAX_POSSIBLE_OPTIONS) {
        throw new BusinessException('Max options amount exceeded');
    }

    $poll->options()->create(...);

    $this->dispatcher->dispatch(new PollOptionAdded($poll));
}
}

final class SomeListener implements ShouldQueue
{
    public function handle(PollOptionAdded $event)
    {
        // ...
        foreach($event->getPoll()->options as $option)
        {...}
    }
}
```

Here is the trap. When the service class checks the options count, it gets the fresh options of the poll. Then it adds the new option to the poll by **$poll->options()->create(…)**; Then the listener asks **$event->getPoll()->options** and gets an old copy of options without the newly added one. This is an Eloquent behaviour, which has two different interfaces to relations. **options()** method and **options** pseudo-field. So, to provide entities to events, the developer has to pay attention to the consistency of entities. For this case:

```
$poll->load('options');
```

should be run before the event firing. The same loading should be called to each relation which can be changed by an action. That’s why for Eloquent entities I recommend just providing the id, not the whole entity. Listeners will always fetch a fresh entity by id.

9. Unit testing

> 100% coverage should be a consequence, not the aim.

First steps

You have perhaps heard about unit testing. It's quite popular nowadays. I often talk with developers who don't start to write any features without writing a unit test for it. TDD guys. It's very difficult to start writing unit tests for your code, especially if you're using a framework like Laravel. Unit tests are one the best indicators for a project's code quality. Frameworks like Laravel are trying to make development as rapid as possible, and allow you to cut corners in many places. High-coupled and low-cohesive code is the usual price for that. Entities hardwired to the database, classes with multiple dependencies which are very hard to find(Laravel facades). In this chapter I'll try to test the Laravel application code and show the main difficulties, but let's start from the very beginning:

Pure function - the function, with a result depending **only** on given parameters, with values only used for reading.

Examples:

```
function strpos(string $needle, string $haystack)
function array_chunk(array $input, $size, $preserve_keys = null)
```

Pure functions are very simple and predictable. Their unit tests are the easiest and simplest unit tests ever.
Our first example will also be a pure function. It can be a function or a method of the stateless class. So, starting by unit tests:

```
function cutString(string $source, int $limit): string
{
    return ''; // some dummy code
}

class CutStringTest extends \PHPUnit\Framework\TestCase
{
```

```
    public function testEmpty()
    {
        $this->assertEquals('', cutString('', 20));
    }

    public function testShortString()
    {
        $this->assertEquals('short', cutString('short', 20));
    }

    public function testCut()
    {
        $this->assertEquals('long string shoul...',
            cutString('long string should be cut', 20));
    }
}
```

I used PHPUnit framework to test this function. The function name isn't the best, but just by reading its tests, you can easily understand what exactly this function should do. Unit tests might be good documentation for your code. If I run my tests, the PHPUnit output will be:

```
Failed asserting that two strings are equal.
Expected :'short'
Actual   :''

Failed asserting that two strings are equal.
Expected :'long string shoul...'
Actual   :''
```

Obviously, our function doesn't do it's job. It's time to implement:

```
function cutString(string $source, int $limit): string
{
    $len = strlen($source);

    if($len < $limit) {
        return $source;
    }

    return substr($source, 0, $limit-3) . '...';
}
```

PHPUnit output after that:

```
OK (3 tests, 3 assertions)
```

Good. The unit test class contains the requirements to function:

- For an empty string, the result also should be empty.
- For strings that don't exceed the limit, the result should be the same.
- For a string longer than the limit, the result should be cut to the needed limit with 3 dots.

Successful tests mean that our function satisfies its requirements. But that's not true! I've made a little mistake and the function works incorrectly if the source string length equals the limit. Good habit: if a bug was found, a test that reproduces this bug should be created first. Anyway, we should check whether this bug was fixed or not and a test is the best place for that check. New test methods:

```
class CutStringTest extends \PHPUnit\Framework\TestCase
{
    // Old tests here

    public function testLimit()
    {
        $this->assertEquals('limit', cutString('limit', 5));
    }

    public function testBeyondTheLimit()
    {
        $this->assertEquals('beyondl...',
                            cutString('beyondlimit', 10));
    }
}
```

testBeyondTheLimit is ok, but **testLimit** fails:

```
Failed asserting that two strings are equal.
Expected :'limit'
Actual   :'li...'
```

Fix is simple: just replace < by <=

```
function cutString(string $source, int $limit): string
{
    $len = strlen($source);

    if($len <= $limit) {
        return $source;
    }

    return substr($source, 0, $limit-3) . '...';
}
```

Run tests immediately:

```
OK (5 tests, 5 assertions)
```

Awesome. Checking these boundary values (0, **$limit** length, **$limit**+1 length, etc.) is an important part of testing. A lot of mistakes happen there.

When I was implementing the cutString function, I thought the source string length would be used twice and saved it to a variable. Now I can remove it.

```php
function cutString(string $source, int $limit): string
{
    if(strlen($source) <= strlen($source)) {
        return $source;
    }

    return substr($source, 0, $limit-3) . '...';
}
```

And again: run tests! I did a refactoring and could have broken something. Better to check it as soon as possible. This feature increases productivity a lot. With good tests, almost every mistake will be caught immediately after making it, when the developer can fix it very quickly.

I put all my attention on the main functionality, but forgot about pre-conditions. Obviously, **$limit** parameter value will never be too small in a real project, but good function design assumes checking this value also:

```php
function cutString(string $source, int $limit): string
{
    if($limit < 5) {
        throw new InvalidArgumentException(
            'The limit is too low');
    }

    if(strlen($source) <= strlen($source)) {
        return $source;
    }

    return substr($source, 0, $limit-3) . '...';
}

class CutStringTest extends \PHPUnit\Framework\TestCase
{
    //...

    public function testLimitCondition()
    {
        $this->expectException(InvalidArgumentException::class);

        cutString('limit', 4);
    }
}
```

Testing stateful classes

Pure functions are great, but there are a lot of things in the real world that can’t be described exclusively with them. Objects can have a **state**. Unit testing of these classes becomes more complex. There is a recommendation for the test method structure to have 3 steps:

1. **initialize** the object and its state
2. **run** the tested action

3. **check** the result

I've also heard about the AAA pattern: Arrange, Act, Assert, which describes the same steps.

Pure function testing also has these steps, but usually all of them are placed in one line. Here is a simple example with an imaginary **Post** entity(it's not an Eloquent entity). It can only be created with a non-empty title, but the body can be empty. If we want to publish this post, the body also should be non-empty.

```
class Post
{
    /** @var string */
    public $title;

    /** @var string */
    public $body;

    /** @var bool */
    public $published;

    public function __construct(string $title, string $body)
    {
        if (empty($title)) {
            throw new InvalidArgumentException(
                'Title should not be empty');
        }

        $this->title = $title;
        $this->body = $body;
        $this->published = false;
    }

    public function publish()
    {
        if (empty($this->body)) {
            throw new CantPublishException(
                'Cant publish post with empty body');
        }

        $this->published = true;
    }
}
```

The Post class constructor is a pure function, so tests for it look similar to the previous example, and the **initialize** and **run** steps are located in one line:

```
class CreatePostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulCreate()
    {
        // initialize and run
        $post = new Post('title', '');
```

```
        // check
        $this->assertEquals('title', $post->title);
    }

    public function testEmptyTitle()
    {
        // check
        $this->expectException(InvalidArgumentException::class);

        // initialize and run
        new Post('', '');
    }
}
```

Otherwise, the **publish** method depends on the post's current state and its tests have more transparent steps:

```
class PublishPostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulPublish()
    {
        // initialize
        $post = new Post('title', 'body');

        // run
        $post->publish();

        // check
        $this->assertTrue($post->published);
    }

    public function testPublishEmptyBody()
    {
        // initialize
        $post = new Post('title', '');

        // check
        $this->expectException(CantPublishException::class);

        // run
        $post->publish();
    }
}
```

In case of testing exceptions, the **check** step, which is usually the last one, should be before the **run** step. Testing stateful classes is a bit more complex than testing pure functions, and a developer who writes a test should keep the entity's state in mind.

Testing classes with dependencies

One of the most important characteristics of unit testing is testing a unit in isolation. The unit should be isolated from the rest of the world. It guarantees that the test will test only this unit. The test can only fail for 2 reasons: wrong

test or wrong unit code. Isolation gives us simplicity and performance. True unit tests run very quickly because they usually don't make heavy I/O operations like database queries or API calls. When a class depends on some other functionality, the test should somehow provide it.

Real class dependencies

In the DI chapter, I was talking about two cases with interfaces:

1. there is an interface and more than one implementations.
2. there is an interface and only one implementation.

For the second case, I suggested to not create an interface. I want to analyze this case now. What dependency can be implemented in only one way? All I/O related dependencies, such as API calls, file operations, and database queries always have other possible implementations (with another driver, decorator pattern, etc.). Sometimes a class contains some big computations and a developer decides to move it to its own class. This new class becomes a dependency for the old one. In that case, another implementation for this class is hard to imagine. This is a great time to talk about encapsulation and why unit testing is called "**unit** testing" (not "**class** testing").

Here is an example of the described case. **TaxCalculator** was moved to its own class from **OrderService**.

```
class OrderService
{
    /** @var TaxCalculator $taxCalculator */
    private $taxCalculator;

    public function __construct(TaxCalculator $taxCalculator)
    {
        $this->taxCalculator = $taxCalculator;
    }

    public function create(OrderCreateDto $orderCreateDto)
    {
        $order = new Order();
        //...
        $order->sum = ...;
        $order->taxSum = $this->taxCalculator
                            ->calculateOrderTax($order);
        //...
    }
}
```

But if we look at this **OrderService** class, we can see that **TaxCalculator** doesn't look like its dependency. It doesn't look like something from the

outside world that **OrderService** needs to work. It looks like a part of the **OrderService** class.

OrderService here is a **unit**, which contains not only the **OrderService** class, but also a **TaxCalculator** class. **TaxCalculator** should be an inner dependency, not outer.

```
class OrderService
{
    /** @var TaxCalculator $taxCalculator */
    private $taxCalculator;

    public function __construct()
    {
        $this->taxCalculator = new TaxCalculator();
    }

    //...
}
```

Unit tests can now test the **OrderService** class without providing the **TaxCalculator** dependency. In case of requirement changes when **TaxCalculator** becomes an outer dependency (some parameters needed for calculation could be moved to the database), it can easily be used as a public dependency, provided as a constructor parameter. Only the constructor code and some tests will be changed.

Unit is a very wide concept. In the beginning of this chapter, we were testing a unit which contained only one little function, **cutString**. In some cases, units can contain several classes. Programming objects inside a unit should be focused on one responsibility, in other words, have strong cohesion. This is what the **Single Responsibility Principle** from SOLID is all about.

When the methods of a class are totally independent from each other, the class is not a unit. Each method in the class is a unit by itself. If you’re going to unit test this class, it may be better to extract the methods to their own classes (do

you remember that I prefer *Command classes for each application layer action instead of *Service classes?). It can simplify the unit testing.

Stubs and fakes

Usual public dependency has an interface and some implementation(s) that are used in the real app. Using real implementations during unit testing is not a good idea. Usually they do some I/O operations or have dependencies which do it. It is usually unacceptable and also makes unit tests too slow. Unit tests should be fast because they will be executed very often. Slow unit test execution can decrease developers' productivity a lot. The most natural solution is to create a dummy implementation of the needed interface and provide it to the class on the unit test. Returning to the previous example, imagine that a requirement really changed and **TaxCalculator** now has an interface with some implementation.

```
interface TaxCalculator
{
    public function calculateOrderTax(Order $order): float;
}

class OrderService
{
    /** @var TaxCalculator $taxCalculator */
    private $taxCalculator;

    public function __construct(TaxCalculator $taxCalculator)
    {
        $this->taxCalculator = $taxCalculator;
    }

    public function create(OrderCreateDto $orderCreateDto)
    {
        $order = new Order();
        //...
        $order->sum = ...;
        $order->taxSum = $this->taxCalculator
                            ->calculateOrderTax($order);
        //...
    }
}

class FakeTaxCalculator implements TaxCalculator
{
    public function calculateOrderTax(Order $order): float
    {
        return 0;
    }
}

class OrderServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreate()
    {
        $orderService = new OrderService(new FakeTaxCalculator());
```

```
        $orderService->create(new OrderCreateDto(...));

        // some assertions
    }
}
```

Works! These classes are called **fakes**. Unit test libraries can create dummy implementations on the fly. The same test using PhpUnit's createMock method:

```
class OrderServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreate()
    {
        $stub = $this->createMock(TaxCalculator::class);

        $stub->method('calculateOrderTax')
            ->willReturn(0);

        $orderService = new OrderService($stub);

        $orderService->create(new OrderCreateDto(...));

        // some assertions
    }
}
```

They are convenient if you need a simple implementation once. If there are a lot of usages of **TaxCalculator**, the fake is the more preferable solution. Mock libraries can create stubs not only for interfaces, but for real classes too, which can be useful for legacy projects or for projects with lazy developers. :)

Mocks

Sometimes a developer wants to test whether this stub method called and which parameters were provided there.

Actually, it may not be the best idea to check each dependency's method call. In that case, a unit test knows a lot about how this unit works. As a consequence, these unit tests usually become too easy to break. A little refactoring and the unit test fails. If it happens too often, the team can just forget about unit testing. It's called "white box" testing. The "black box" testing style tries only test inputs and outputs of the unit without trying to check how it works inside. Obviously, black box tests are more stable.

This checking can be implemented in the fake class, but it is not trivial and you definitely don't want to do it for each dependency. Mocking libraries provide this functionality.

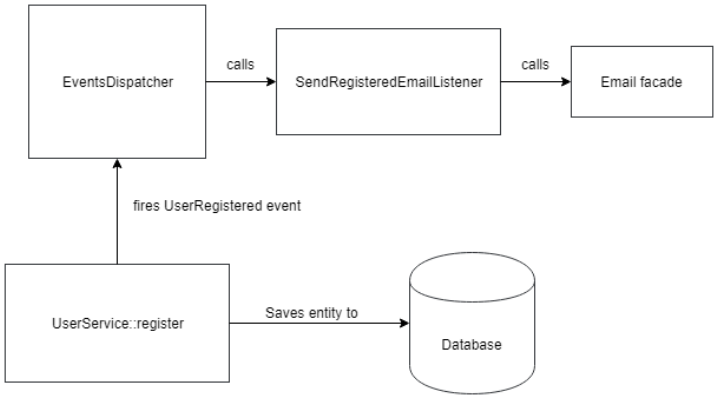
```php
class OrderServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreate()
    {
        $stub = $this->createMock(TaxCalculator::class);

        // Configure the stub.
        $stub->expects($this->once())
            ->method('calculateOrderTax')
            ->willReturn(0);

        $orderService = new OrderService($stub);

        $orderService->create(new OrderCreateDto(...));

        // some assertions
    }
}
```

Now the test checks that during the OrderService::create method execution, **TaxCalculator::calculateOrderTax** was called exactly once. Mock methods have a lot of features for parameter value checking, returning values configuring, throwing exceptions, etc. I don't want to focus on it in this book. Fakes, stubs and mocks have a common name - test doubles, which means objects which stand in for real objects in test. They can be used not only in unit tests but in integration tests also.

Software testing types

People invented a ton of possible types of software testing. Security testing to find possible vulnerabilities. Performance testing to check application performance. In this chapter, we will focus on application correctness testing. We already took a look at **unit testing**, where units were tested in isolation. **Integration testing** tests several units working together.

Example: ask **UserService** to register a new user and check that a new row was saved to the database, the needed event(**UserRegistered**) was generated and an email for this user was sent(at least the framework was asked to send it).



Functional testing is about checking the whole app to accept its functional requirements.

Example: Requirement about creating some entity (this process can be described in detail in QA's documents). Test opens the browser, goes to the specified page, fills inputs, presses the **Create** button and checks that the needed entity was created by finding it on another page.

Laravel testing

Laravel (current version is 5.8) provides some tools to simplify various kinds of testing.

Functional testing

Tools for HTTP testing, browser testing and console testing make functional testing in Laravel very convenient, but as usual I don't like the examples in documentation. One of them, but a little bit changed:

```
class ExampleTest extends TestCase
{
    public function testBasicExample()
    {
        $response = $this->postJson('/users', [
            'name' => 'Sally',
            'email' => 'sally@example.com'
        ]);

        $response
            ->assertOk()
            ->assertJson([
                'created' => true,
            ]);
```

```
    }
}
```

This test just tests that **POST /user** request returns a successful result. It looks incomplete. The test should verify that the entity was really created. How? First answer: just make a query for the database and check it. Example from documentation again:

```
class ExampleTest extends TestCase
{
    public function testDatabase()
    {
        // Make call to application...

        $this->assertDatabaseHas('users', [
            'email' => 'sally@example.com'
        ]);
    }
}
```

Good. Let’s write another test the same way:

```
class PostsTest extends TestCase
{
    public function testDelete()
    {
        $response = $this->deleteJson('/posts/1');

        $response->assertOk();

        $this->assertDatabaseMissing('posts', [
            'id' => 1
        ]);
    }
}
```

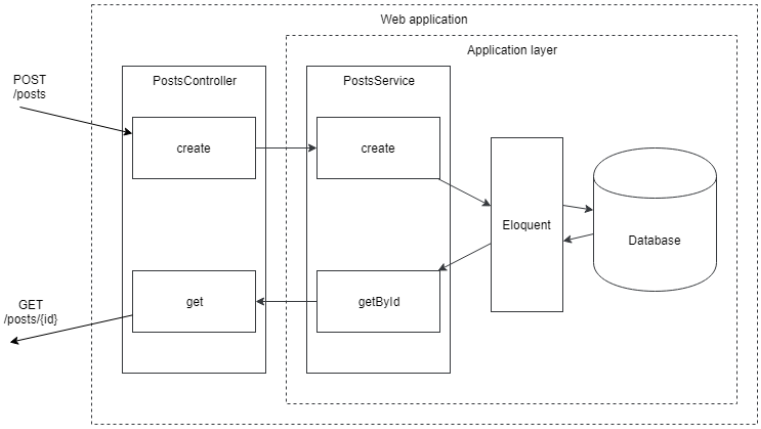
And here is a little trap. The same trap as we met in the **Validation** chapter. Just by adding the **SoftDeletes** trait to the **Post** class, this test will be broken. Application requirements haven’t changed, application from the user’s point of view works absolutely the same. Functional tests should not be broken in that case. Tests which make a request to the application and go to the database to check the result of actions aren’t true functional tests. They know too much about how the application works, how it stores the data, and which table and fields are used. It is an another example of white box testing.

As I already mentioned, functional testing is about checking the whole app to accept it’s functional requirements. Functional requirements are not about the database, they are about the whole application. True functional tests should work only outside the app.

```php
class PostsTest extends TestCase
{
    /**
     * A basic test example.
     *
     * @return void
     */
    public function testCreate()
    {
        $response = $this->postJson('/api/posts', [
            'title' => 'Post test title'
        ]);

        $response
            ->assertOk()
            ->assertJsonStructure([
                'id',
            ]);

        $checkResponse = $this->getJson(
            '/api/posts/' . $response->getData()->id);

        $checkResponse
            ->assertOk()
            ->assertJson([
                'title' => 'Post test title',
            ]);
    }

    public function testDelete()
    {
        // Some initialization here to create a post with id=$postId
        // Make sure that post exists in application
        $this->getJson('/api/posts/' . $postId)
            ->assertOk();

        // Delete post command
        $this->jsonDelete('/posts/1')
            ->assertOk();

        // Check that post doesn't exist in application
        $this->getJson('/api/posts/' . $postId)
            ->assertStatus(404);
    }
}
```

At delete test, we don’t care if it is soft deleted, or deleted by ‘delete’ sql query. Functional test just checks that the application behaves as expected. Expected behavior for deleted entity - it isn’t shown anywhere in the app and the test should check it.

Schema of data flow for “create post” and “get post” request processing:



What functional tests should see:

Facades mocking

Laravel also suggests to use their convenient implementation of Service Locator pattern - Laravel facades. I always say “Laravel facade”, because there is a **Facade** structural design pattern and people can misunderstand which facade is used. In one project, two developers said that I don’t understand how Laravel facades work and I had to write comments like this:

```
/**
 * It is NOT a Laravel facade.
 * It's an implementation of a Facade pattern from GoF.
 */
final class SomeFacade
{
```

```
    //...
}
```

Laravel not only suggests to use them but also provides good tools for testing the code, which uses facades. Let’s try to write one of the previous examples with facades and test it:

```
class Poll extends Model
{
    public function options()
    {
        return $this->hasMany(PollOption::class);
    }
}

class PollOption extends Model
{
}

class PollCreated
{
    /** @var int */
    private $pollId;

    public function __construct(int $pollId)
    {
        $this->pollId = $pollId;
    }

    public function getPollId(): int
    {
        return $this->pollId;
    }
}

class PollCreateDto
{
    /** @var string */
    public $title;

    /** @var string[] */
    public $options;

    public function __construct(string $title, array $options)
    {
        $this->title = $title;
        $this->options = $options;
    }
}

class PollService
{
    public function create(PollCreateDto $dto)
    {
        if(count($dto->options) < 2) {
            throw new BusinessException(
                "Please provide at least 2 options");
        }

        $poll = new Poll();
```

```
        \DB::transaction(function() use ($dto, $poll) {
            $poll->title = $dto->title;
            $poll->save();

            foreach ($dto->options as $option) {
                $poll->options()->create([
                    'text' => $option,
                ]);
            }
        });

        \Event::dispatch(new PollCreated($poll->id));
    }
}

class PollServiceTest extends TestCase
{
    public function testCreate()
    {
        \Event::fake();

        $postService = new PollService();
        $postService->create(new PollCreateDto(
            'test title',
            ['option1', 'option2']));

        \Event::assertDispatched(PollCreated::class);
    }
}
```

- **Event::fake()** call transforms **Event** facade to mock object.
- **PostService::create** creates a poll with options, saves it to the database and dispatches a **PollCreated** event.
- **Event::assertDispatched** checks that the needed event was dispatched.

Disadvantages I see here:

- It’s not a unit test. The **Event** facade was mocked, but database was not. Real table rows are inserted during test execution. To make this test more clear, the **RefreshDatabase** trait is usually used to re-create the database for each test. It’s very slow. One test can be executed this way in a reasonable time, but hundreds of tests will be executed over several minutes. It’s not acceptable.
- The database is touched during this test. That means the database in an ideal case should be the same as in production. Sometimes it’s not possible and issues happen locally, but not in production or visa versa, which can make these tests almost useless. Also, some developers store their projects in virtual machines, but want to run tests under a host machine, which doesn’t have the database and other environment installed.
- Test checks only event dispatching. To check database operations, the developer has to use methods like **assertDatabaseHas** or something like

PollService::getById, which makes the test look like a functional test for the application layer.
- **PollService** class dependencies aren't described explicitly. To check what it needs to work, we have to scan the whole source. It makes **PollService** test writing more complicated. Even worse, if some new dependency will be added to this class, tests will continue working using a real implementation of this dependency: real API calls, real file creating, etc.

I call this "forced integration testing". Developer wants to create a unit test, but the code is so high coupled, so hardwired with framework (in this example with Service locator and Eloquent)… I'm going to try to decouple the code from the framework in the next section.

Application layer unit testing

Decouple from Laravel facades

To test the **PollService::create** method in isolation, I have to decouple it from Laravel facades and the database(Eloquent). First task is simple, as Laravel facades are easy to change by DI.

- **Event** facade represents **IlluminateContractsEventsDispatcher** interface.
- **DB** facade - **IlluminateDatabaseConnectionInterface**.

Well, the last one isn't correct. **DB** facade represents **IlluminateDatabaseDatabaseManager** which has this "magic":

```
class DatabaseManager
{
    /**
     * Dynamically pass methods to the default connection.
     *
     * @param  string  $method
     * @param  array   $parameters
     * @return mixed
     */
    public function __call($method, $parameters)
    {
        return $this->connection()->$method(...$parameters);
    }
}
```

As you see, Laravel uses PHP magic a lot and sometimes doesn't respect OOP principles. Thanks to **barryvdh/laravel-ide-helper** package, which helps to

find the real implementors of some Laravel facade methods.

```
class PollService
{
    /** @var \Illuminate\Database\ConnectionInterface */
    private $connection;

    /** @var \Illuminate\Contracts\Events\Dispatcher */
    private $dispatcher;

    public function __construct(
        ConnectionInterface $connection, Dispatcher $dispatcher)
    {
        $this->connection = $connection;
        $this->dispatcher = $dispatcher;
    }

    public function create(PollCreateDto $dto)
    {
        if(count($dto->options) < 2) {
            throw new BusinessException(
                "Please provide at least 2 options");
        }

        $poll = new Poll();

        $this->connection->transaction(function() use ($dto, $poll) {
            $poll->title = $dto->title;
            $poll->save();

            foreach ($dto->options as $option) {
                $poll->options()->create([
                    'text' => $option,
                ]);
            }
        });

        $this->dispatcher->dispatch(new PollCreated($poll->id));
    }
}
```

Good. For the **ConnectionInterface** test double I can create a **FakeConnection** class. Laravel's **EventFake** class, which was used to replace **Event** after **Event::fake()** call, can be used independently.

```
use Illuminate\Support\Testing\Fakes\EventFake;
//...

class PollServiceTest extends TestCase
{
    public function testCreatePoll()
    {
        $eventFake = new EventFake(
            $this->createMock(Dispatcher::class));

        $postService = new PollService(
            new FakeConnection(), $eventFake);

        $postService->create(new PollCreateDto(
            'test title',
```

```
            ['option1', 'option2']));

        $eventFake->assertDispatched(PollCreated::class);
    }
}
```

The test looks almost the same as with facades, but now it is more strict for the **PollService** class. Each added dependency has to be added to the tests, also. Does it? Actually, if some other developer adds a Laravel facade using the **PollService** class, nothing happens and the test will work as usual, even calling this facade! It happens because by default, Laravel provides its own base **TestCase** class, which sets up the Laravel environment.

```
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;
}
```

That's why I usually change it to a pure PhpUnit **TestCase** class:

```
abstract class TestCase extends \PHPUnit\Framework\TestCase
{
}
```

Now if someone adds **SomeFacade** facade call, the test will fail:

```
Error : Class 'SomeFacade' not found
```

Well, I decoupled **PollService** from Laravel facades.

Decouple from database

Decoupling from the database is more complex. Let's create a repository class to use instead of Eloquent actions that touch the database.

```
interface PollRepository
{
    //... some other actions

    public function save(Poll $poll);

    public function saveOption(PollOption $pollOption);
}

class EloquentPollRepository implements PollRepository
{
    //... some other actions

    public function save(Poll $poll)
    {
```

```
        $poll->save();
    }

    public function saveOption(PollOption $pollOption)
    {
        $pollOption->save();
    }
}

class PollService
{
    /** @var \Illuminate\Database\ConnectionInterface */
    private $connection;

    /** @var PollRepository */
    private $repository;

    /** @var \Illuminate\Contracts\Events\Dispatcher */
    private $dispatcher;

    public function __construct(
        ConnectionInterface $connection,
        PollRepository $repository,
        Dispatcher $dispatcher)
    {
        $this->connection = $connection;
        $this->repository = $repository;
        $this->dispatcher = $dispatcher;
    }

    public function create(PollCreateDto $dto)
    {
        if(count($dto->options) < 2) {
            throw new BusinessException(
                "Please provide at least 2 options");
        }

        $poll = new Poll();

        $this->connection->transaction(function() use ($dto, $poll) {
            $poll->title = $dto->title;
            $this->repository->save($poll);

            foreach ($dto->options as $optionText) {
                $pollOption = new PollOption();
                $pollOption->poll_id = $poll->id;
                $pollOption->text = $optionText;

                $this->repository->saveOption($pollOption);
            }
        });

        $this->dispatcher->dispatch(new PollCreated($poll->id));
    }
}

class PollServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreatePoll()
    {
        $eventFake = new EventFake(
            $this->createMock(Dispatcher::class));

        $repositoryMock = $this->createMock(PollRepository::class);
```

```
        $repositoryMock->method('save')
            ->with($this->callback(function(Poll $poll) {
                return $poll->title == 'test title';
            }));

        $repositoryMock->expects($this->at(2))
            ->method('saveOption');

        $postService = new PollService(
            new FakeConnection(), $repositoryMock, $eventFake);

        $postService->create(new PollCreateDto(
            'test title',
            ['option1', 'option2']));

        $eventFake->assertDispatched(PollCreated::class);
    }
}
```

This is a correct unit test. **PollService** was tested in isolation, without Laravel environment. But why am I not very happy about this? The reasons are:

- I had to create a **Repository** abstraction just to make my code testable. **PollService** code without it looks more readable, which is important. It looks like a **Repository** pattern, but it's not. It just substitutes all Eloquent database operations. If the **Poll** entity will have more relations, the repository class should have a **save%RelationName%** method for each of them.
- Almost all Eloquent methods are restricted. Yes, they will work correctly in real request processing, but not in unit tests. Over time, developers will ask "do we really need these unit tests?"
- On the other hand, unit tests are very complicated. This example only checks that the poll with two options was correctly created. Even for this simple example I had to use two fakes and one mock.
- Each new dependency addition or removal breaks the tests. It makes unit test supporting harder.

It's hard to measure, but I think the benefits of these unit tests are less than the time spent to write/support them and reduces the readability of the application layer. In the beginning of this chapter I said that "unit tests are one the best indicators for projects code quality". If the code is hard to test, it means the code is low cohesive and/or high coupled. **PollService** is low cohesive. It violates the Single Responsibility Principle. It contains the core logic (checking options count and creating poll with options) and the application logic (database transactions, events dispatching, bad words filter in previous edition, etc.). This

can be solved by separating Domain and Application layers. The next chapter is about that.

Application testing strategy

In this section I don’t want to speak about big companies, which usually create the testing strategy right after starting the new project. I want to talk about little projects that begin to grow. First, the project was tested just team members using it. Over time, the project owner, manager or developer opens the app, does some actions and checks the correctness or how beautiful the UI is. It’s a non-automated Functional testing without any strategy.

Then in the same stage (usually after some painful bugs in the production), the team understands they need to change something.

First, the obvious solution is to hire a manual software tester. He can describe the main use cases for the app and after each app update go through these use cases and check that the application meets the requirements (it’s called **regression testing**).

Then, if the application continues growing, the number of use cases also grows. At the same time, the team wants to make updates more often and it becomes totally impossible to manually test the application for each update. The solution is to write automated functional tests. Use cases written in the first stage can be converted to test scenarios for Selenium or other tools for functional testing.

Functional testing is the most important testing from a client’s/product owner’s point of view.

What about unit tests? Yes, they can help to check very special cases that are hard to check in functional testing, but the main benefit from them - they should help us to write the code. Do you remember the example with the cutString function in the beginning of this chapter? It was much easier to write the code, which meets its requirements with unit tests. Writing unit tests should be easy. They shouldn’t be a heavy burden of the project, which is not very stable and always takes time to support. If you have some pure functions or some stateful classes, unit tests can help you to create them with the needed quality quicker than doing it without unit tests.

The application layer unit tests are too difficult to create and support. “Forced integration” tests are easier to create but might be very unstable. If your project has the core logic, which you definitely want to check how it will behave in boundary cases, it may be a big hint for you. Maybe the project core logic is already big and complex enough to separate it from Application layer to its own Domain layer.

10. Domain layer

> private $name, getName() and setName($name) is NOT an encapsulation!

When and why?

Complex web applications can be classified into two classes:

- Complex application logic and average (or low) core logic. Example: content projects. Application logic has SEO actions, integrations with different API's, authorization, searching the content. Core logic is very simple in comparison with application logic.
- Complex core logic and complex or not very complex application logic. Example: games, enterprise apps.

It's easy to predict that I suggest to extract core logic (which is usually called Domain or Business logic) to its own layer for a second class of complex applications (for other apps it also might be useful, but it's not easy to prove and depends on many factors). This moving to its own layer

might be very difficult, especially for Laravel projects, because Eloquent is hardwired with database, so creating a pure domain layer with Eloquent entities is impossible. I'll try to discuss the reasons for creating a Domain layer. Each of them is not a super strong argument, but together they can prove the need for extracting a Domain layer for some apps.

Unit testing

In the previous chapter I've found that unit testing application logic might be very difficult. So for applications with not very large Domain logic, nobody usually wants to unit test the application layer. Functional tests are much more important for this kind of app.

On the other hand, complex logic is hard to test with functional tests. Functional tests are very expensive. It takes some time to write and, more important, it takes a lot of time to run, because it needs all application environments installed and involved for the test run. Thousands of unit tests can be run in several seconds. Thousands of functional tests could run for hours.

Using unit testing for complex Domain logic can increase team productivity, but as we saw in the previous chapter, testing Domain logic inside the Application layer is definitely not the good idea. Each unit test should provide test doubles for application layer dependencies, just to check the business logic.

Implementing the business logic in a separated layer makes the unit testing much easier.

Easier code maintenance

Implementing two different types of logic (application and business) in the same unit violates the Single Responsibility Principle. Punishment comes quickly. This code easily becomes hard to support. The number of copy-pasted code blocks grows because refactoring is difficult. Each try to extract logic encounters resistance: two different logics intertwined, and extracting something requires pre-separating them.

Extracting business logic to its own layer allows the developer to concentrate similar logic in one place, which always makes refactoring and maintenance of this section of code easier.

Active record and data mapper

Eloquent is an implementation of the Active Record pattern. Eloquent entity represents the database table row and is inseparable from the database. Each time I write code of an Eloquent entity class, I keep in mind that $this->posts is not just a collection of posts. It's a database relation, a set of related table rows. But I can't just add a new post there, I should do something like $this->posts()->create(…);

This code can't be unit tested. I also can't write without thinking about the database. The cognitive load for writing the code becomes higher and higher when the business logic grows.

Data Mapper libraries allow to write business entities without thinking about the database. Entities there are usual classes and they can be used even without the database. Unit testing them is very easy. Unit tests just work with natural entities, which only implement the business logic. When a developer wants to store this entity in the database, it asks Data Mapper library to **persist** entity and the library does this using some meta-information about how each class and field should be persisted in database. I saw two Data Mapper ORM libraries working in Laravel projects: Doctrine and Analogue. I'll use the first one for examples. The following "reasons" will describe advantages of using Data Mapper entities instead of Active record entities.

High cohesion for Domain logic

Let's return to an example with poll entity. Poll is very simple entity that contains question text and possible answers (options). Obvious condition: there should be always at least two options. As you see, "create poll" action checks that before creating poll object. This condition also makes the **PollOption** entity not independent! Application can't just delete a poll option. "delete poll option" action should check how many options the poll has. In case of only two options, the option can't be deleted because poll

can’t be left with only one option. So, knowledge about the minimum of two options for a poll is implemented in two application layer actions: “create poll” and “delete poll option”. Cohesion of this knowledge is low! It’s implemented in two methods of usually different classes.

It happened because **Poll** and **PollOption** are not independent entities. It is one unit - **Poll**, which contains options. This knowledge about the minimal number of options should be implemented in one place, in the Poll entity.

Well, this simple example can’t prove the importance of working with entities as a unit when they are real units. Let me imagine something more complex - an implementation of a Monopoly game. Everything in this game is one big unit. Players, their properties, their money, their position on the board. All of this represents the current state of the game. A player makes the move and the next state completely depends on the current state. If he steps on someone’s property, he should pay. If he has enough money, he pays. If not, he should earn the money or retreat. If the property is free, he can buy it. If he doesn’t have enough money, an auction between other players begins.

Monopoly is a good example of a complex unit. Its implementation in different application layer actions will contain a lot of duplicated logic. Creating a Monopoly game unit in the Domain layer allows the developer to concentrate the game logic in one place and check a lot of possible cases by unit testing it.

Move focus from database to domain.

The app, especially when complex, is not a mediator from UI and database. Single player games played in a local computer don’t persist its own state after each action. Game objects just live in memory, without any persisting. Only if the user asks to save the game, then it persists the state to the file. This is how an application should work. Web applications have to persist the state after each request, not because it’s more convenient. It’s a necessary evil. In an “ideal” world with applications, which are written once and never updated, working 100% stable in a 100% stable server with unlimited memory, the best solution is to keep everything in application memory, without persisting anything to databases. This idea looks crazy,

especially for PHP developers (php script dies after each request), but it makes sense. From this point of view, it is much better to have usual objects that contain the business logic and have no idea about such strange things - database tables. This will help developers to focus on core logic, not on infrastructure issues.

Example project: freelance market. Some freelance market apps have different registration/login parts for clients (users who post jobs) and freelancers. Other apps have common registration and allow users to act as a client and as a freelancer. First ones usually have two different tables, like **clients** and **freelancers**. Second ones only have a **users** table.

In a project with Eloquent-like libraries, developers have to choose entities depending on database structure. For Job actions: **Client** & **Freelancer** or **User** & **User**. But for core logic and the language we use to describe it, it should not depend on something not in the domain. When I describe job posting and applying processes, I want to use entities with natural names. With names that are understandable not only for developers, but also for anybody who knows the domain. I mean **Client** and **Freelancer**.

I also want to use meaningful Value objects in the domain. If a client has an email, I want to use **Email** object, not a string. If some entity has an address value, I want to work with **Address** object, not with five string values.

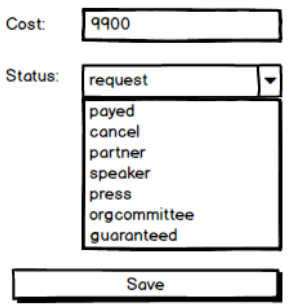
Data Mapper libraries usually have support for flexibly mapping entities and value objects to corresponding tables and fields.

Move focus from UI to domain

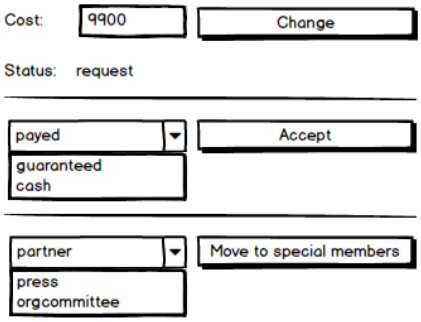
Some developers start their projects from creating a migration to database. Others start a projects from building a user interface (UI). All these factors affect our code.

Once I helped to create a web application for a developers' conference. Just a website with a schedule, making orders and payment. In the beginning we had an easy admin panel with CRUD-like forms for orders. Simple form with customer name, order cost, order status (request, payed, cancel) and some details. When we decided to track all attendees in our app, the process

became a bit complicated. Conference partners who wanted to participate had to create an order, tell the order number to the organizer, who changes the order cost to zero and status to… payed? No. We had to separate these members from others, so a new order status appeared - “partner”. Then “speaker”, “press”, etc. Another status, “guaranteed”, appeared for cases when some business couldn’t pay on time, but guaranteed to do it later. The UI form continued to be very “simple” but the cognitive load for working with it was growing.



It happens very often. The customer, product owner or developer doesn’t want to change the old UI, even if requirements have changed a lot. Application code tries to satisfy the UI requirements, which become incorrect. Incorrect UI results in incorrect domain, the code’s cognitive load also grows, and the project easily becomes hard to support. For the conference, we decided to separate special members (speakers, partners, press) and orders. Order status returned to three values: request, cancel and accepted. New field acceptReason explains why this order is accepted: payed, guaranteed, cash (when customer wants to pay by cash on conference day). Yes, we created some new entities, but several simple classes are much better than one very complex class. The same rule goes for UI. When we changed the UI according to our new domain, the organizers weren’t very happy, but a bit later (1-2 days) they understood that it became much more logical and didn’t allow for mistakes.



A well-designed domain allows a developer to build a good UI and database structure. Unit testing can show the flaws of poor-designed domain more quickly than UI/UX analysis.

Entities classes invariant

Class invariant is a condition for an object's state which should always be fulfilled. Each method that changes the object's state (including a constructor) should preserve invariant. Examples: client should always have an email, poll should always have at least two options. The requirement of preserving an invariant after each write-method call forces implementing invariant checks in each of them.

I didn't find a good name for changing state methods, so let me call them "write-methods". "Read-method" just returns some info from a class. It's the same as setter/getters, but works not only for one value.

By the way, the idea of immutable objects makes this task easy. The invariant should only be checked in the constructor. There are no other write-methods. All value objects should be immutable and as you see, the **Email** object only checks the email in the constructor. In case of the **setEmail** method, the check should be duplicated there. So, more easy fulfillment of invariant is one of the advantages of immutable objects.

It's very hard to preserve the invariant in Eloquent entities. Eloquent doesn't allow to control the constructor. **PollOption** objects are not under **Poll** object control. Each part of the application code can just call remove() method of **PollOption** and it will be removed. Yes, this argument can be

easily rejected by advice to not do such a stupid thing like removing options without any checks, but the real domain can be much more complicated and the team can be very large and contain developers with different levels and understanding of domain. If code doesn't allow a developer to make the mistake, obviously it's more stable than code which relies only on smart and high-level developers.

Implementation of Domain layer

Which logic is "business"

Once, I'd found a set of Laravel "best" practices, there were very strange pieces of advice, and one of them was:

"Business logic should be in the service class" with example:

```
public function store(Request $request)
{
    $this->articleService
        ->handleUploadedImage($request->file('image'));

    ....
}

class ArticleService
{
    public function handleUploadedImage($image)
    {
        if (!is_null($image)) {
            $image->move(public_path('images') . 'temp');
        }
    }
}
```

This illustrates how difficult it could be to understand what business logic is. The "business" is the keyword. Commercial development is not for fun. It creates a software that solves business tasks. An e-commerce app works with products, orders, discounts, etc. A blog app works with posts and other content. Images are also content, but for the majority of apps, images are uploaded to local or cloud storage and only the path of it is provided to the main entity (like a blog post). The image itself can be a part of business logic only for a graphical editor or OCR apps.

The image uploading process is not a business logic, it’s an infrastructure logic. Infrastructure code allows the domain layer to live inside the application and satisfy the functional requirements. It contains working with databases, file storage, external APIs, queue engines, etc. When domain logic is extracted from application layer, the last one will contain only an orchestration between infrastructure and domain.

Domain example

Let’s create some domain logic. Freelance market is a good choice.

```
final class Client
{
    /** @var Email */
    private $email;

    private function __construct(Email $email)
    {
        $this->email = $email;
    }

    public static function register(Email $email): Client
    {
        return new Client($email);
    }
}
```

Just a client entity.

Where is the client’s first and last names?

I don’t need it for now. Who knows, the app might be with anonymous clients. If a project owner asks to add names, they will be added.

Why do I need email?

I have to identify clients somehow.

Where is setEmail and getEmail methods?

I’ll add them as soon as I need them.

```
final class Freelancer
{
    /** @var Email */
    private $email;
```

```
    /** @var Money */
    private $hourRate;

    private function __construct(Email $email, Money $hourRate)
    {
        $this->email = $email;
        $this->hourRate = $hourRate;
    }

    public static function register(
        Email $email, Money $hourRate): Freelancer
    {
        return new Freelancer($email, $hourRate);
    }
}
```

A Freelancer entity. Hour rate was added comparing the Customer. Money is the value object used for money representation in domain. What fields will it store? Integer or float with string for currency? Freelancer entity doesn't care about this. Money just represents money. It can do everything the domain needs from it - comparing with another money object, some mathematical calculations.

In the beginning, a project works with one currency and stores money information in one integer field (amount of cents). Later, the project can support other currencies. Some fields will be added to database, and the **Money** class will be changed. Other core logic will stay unchanged because it shouldn't be involved in this change. This is an example of **Information Hiding** principle. The **Money** class provides a stable interface for working with money concept. **getAmount()**:int and **getCurrency()**:string methods aren't good candidates for stable interface. Money class users (I mean the code which uses the class) will know too much about the inner structure and each change of it will result in massive changes in project. **equalTo**(**Money $other**) and **compare**(**Money $other**) methods hide all unneeded information from callers. Information hiding makes interfaces more stable. Less interface change - less pain during code support.

Next, client can post a job. Job has some details, like title, description, estimated budget, but job logic doesn't depend on these values. Freelancer's applying logic will work even without job title. User interface can be changed in the future, some job image can be added. Remembering the Information hiding principle, I hide information of job details in its own value object.

```php
final class JobDescription
{
    // value object.
    // Job title, description, estimated budget, etc.
}

final class Job
{
    /** @var Client */
    private $client;

    /** @var JobDescription */
    private $description;

    private function __construct(Client $client,
        JobDescription $description)
    {
        $this->client = $client;
        $this->description = $description;
    }

    public static function post(Customer $client,
        JobDescription $description): Job
    {
        return new Job($client, $description);
    }
}
```

Okay, the basic entities structure is built. Let’s add some logic. Freelancer can apply for a job. Freelancer’s proposal for job contains cover letter and his current hour rate. He can change his hour rate, but proposal’s hour rate should not be changed in that case.

```php
final class Proposal
{
    /**
     * @var Job
     */
    private $job;

    /**
     * @var Freelancer
     */
    private $freelancer;

    /**
     * @var Money
     */
    private $hourRate;

    /**
     * @var string
     */
    private $coverLetter;

    public function __construct(Job $job,
        Freelancer $freelancer, Money $hourRate, string $coverLetter)
    {
```

```
        $this->job = $job;
        $this->freelancer = $freelancer;
        $this->hourRate = $hourRate;
        $this->coverLetter = $coverLetter;
    }
}

final class Job
{
    //...

    /**
     * @var Proposal[]
     */
    private $proposals;

    protected function __construct(
        Client $client, JobDescription $description)
    {
        $this->client = $client;
        $this->description = $description;
        $this->proposals = [];
    }

    public function addProposal(Freelancer $freelancer,
        Money $hourRate, string $coverLetter)
    {
        $this->proposals[] = new Proposal($this,
            $freelancer, $hourRate, $coverLetter);
    }
}

final class Freelancer
{
    //...

    public function apply(Job $job, string $coverLetter)
    {
        $job->addProposal($this, $this->hourRate, $coverLetter);
    }
}
```

Here is another example of **Information Hiding**. Only the **Freelancer** entity knows about his hour rate. It asks job to add a new proposal with its own hour rate. Each object has minimum necessary information to work. The system built this way is very stable. Each requirement change usually requires changes in 1-2 classes, because the inner structure of any class is hidden.
If something is changed there, but the class interface isn't, there is no need to change other classes. This code is less prone to bugs.

Freelancer can't add another proposal to the same job. He should change the old one instead. Later, in database level, this condition can be duplicated

by a unique index for **job_id** and **freelancer_id** fields in the **proposals** table, but it should be implemented in the domain layer, also. Let's try.

```
final class Proposal
{
    // ..

    public function getFreelancer(): Freelancer
    {
        return $this->freelancer;
    }
}

final class Freelancer
{
    public function equals(Freelancer $other): bool
    {
        return $this->email->equals($other->email);
    }
}

final class Job
{
    //...

    public function addProposal(Freelancer $freelancer,
        Money $hourRate, string $coverLetter)
    {
        $newProposal = new Proposal($this,
            $freelancer, $hourRate, $coverLetter);

        foreach($this->proposals as $proposal) {
            if($proposal->getFreelancer()
                ->equals($newProposal->getFreelancer())) {
                throw new BusinessException(
                    'This freelancer already made a proposal');
            }
        }

        $this->proposals[] = $newProposal;
    }
}
```

I added **equals()** method to **Freelancer** entity. Unique emails is a usual condition for systems, so if two freelancers' entities have the same emails, it's the same freelancer. Class **Job** is beginning to know too much about proposals. If you look inside this foreach structure, there are only **Proposal** class getter calls and working with their results. Martin Fowler named this problem "Feature Envy". The solution is obvious - move this part of code to the **Proposal** class.

```
final class Proposal
{
    //...
```

```
    /**
     * @param Proposal $other
     * @throws BusinessException
     */
    public function checkCompatibility(Proposal $other)
    {
        if($this->freelancer->equals($other->freelancer)) {
            throw new BusinessException(
                'This freelancer already made a proposal');
        }
    }
}

final class Job
{
    /**
     * ...
     * @throws BusinessException
     */
    public function addProposal(Freelancer $freelancer,
        Money $hourRate, string $coverLetter)
    {
        $newProposal = new Proposal($this,
            $freelancer, $hourRate, $coverLetter);

        foreach($this->proposals as $proposal) {
            $proposal->checkCompatibility($newProposal);
        }

        $this->proposals[] = $newProposal;
    }
}
```

Proposal::getFreelancer() method is not used anymore and can be deleted.

Encapsulation is one of the basic principles of object oriented programming, but I often see a wrong interpretation of it. “public **$name** is not an encapsulation, but private **$name** and **getName** and **setName** methods is”. I don’t see the difference between public field and getter+setter methods. They both show the inner structure of this class to client code. It allows the use of values of this class’ fields in other classes. Class becomes a simple storage for some values, not a real owner of them. It’s okay for some classes, like DTO, but not for business entities. Concentrating all logic that works with class fields in this class makes the code highly cohesive. Changes in this logic will touch only this class, not several classes in different parts of the application. Encapsulation, being part of **Information Hiding**, is also a good goal to aim for.

I already wrote some logic but totally forgot about unit testing!

```php
class ClientTest extends TestCase
{
    public function testRegister()
    {
        $client = Client::register(Email::create('some@email.com'));

        // Assert what?
    }
}
```

Client entity doesn't have any getter methods. Unit test can't check anything. PHPUnit will show an error: "This test did not perform any assertions". I can implement a **getEmail()** method and check at least the email, but this method will be executed only in unit tests. It won't be participating in any business operations. I don't want to add the code, which will be executed only in tests, to the domain…

Domain events

It's a good time to remember about domain events! They won't be used only for tests. The application can send an email for the client, after registration. **Client** can fire the **ClientRegistered** event, but here are two problems.

When all business logic was in the Application layer, the action just used a dispatcher object. Domain entity doesn't have any dispatcher objects. The most popular solution is to store generated events in an array inside the entity and clear them each time they will be asked outside.

```php
final class Client
{
    //...

    private $events = [];

    public function releaseEvents(): array
    {
        $events = $this->events;
        $this->events = [];

        return $events;
    }

    protected function record($event)
    {
        $this->events[] = $event;
    }
}
```

Entity can record events by **record** method. **releaseEvents** method returns all previously recorded events and clears the events array. It guarantees that each event won't be dispatched twice.

What should the **ClientRegistered** event contain? I said that I want to use email to identify clients, so **ClientRegistered** event can contain a client's email but it's not a good idea. In the real world, email is not a good identification. Clients can change their emails. Also it is not very effective to use it as a key in the database.

The most popular solution for entity identification is an integer field with auto-incremental value implemented in database. It's simple, convenient, but looks logical only when the domain layer isn't extracted from other code. One of the advantages of using pure domain entities is consistency, but in the period of time between creating and persisting the entity to the database, it will be inconsistent (**id** value is empty). Previously, I mentioned that some heavy requests' executions might be moved to queue. How will the client know where to get a result in that case? Application should generate an **id**, put the task to the queue and return this **id** to the client. **Client** will use this **id** to get the result later.

I still didn't say that I need an id to unit test the entities creation, but I don't like the "unit testing" argument. Inconvenient unit testing can only help to find the problems in the code. If something needs to be added or changed in the code just for unit tests, something is wrong with this code or tests.

Code with providing the id to entities and recording the events:

```
final class Client
{
    private $id;

    /**
     * @var Email
     */
    private $email;

    protected function __construct($id, Email $email)
    {
        $this->id = $id;
        $this->email = $email;
    }

    public static function register($id, Email $email): Client
```

```
    {
        $client = new Client($id, $email);
        $client->record(new ClientRegistered($client->id));

        return $client;
    }
}
```

The **ClientRegistered** event is recorded in named constructor, not in default, because it has the business name “register” and knows that the client really registered. Some clients importing command might be added in the future and the event will be another one.

```
final class Client
{
    public static function importFromCsv($id, Email $email): Client
    {
        $client = new Client($id, $email);
        $client->record(new ClientImportedFromCsv($client->id));

        return $client;
    }
}
```

Id generation

Our code now requires the id provided outside, but how to generate them? The database auto-incremental column does this job flawlessly and it will be hard to replace it. Continuing to use the auto-incremental value in Redis/Memcache is not a good idea, because it adds a new single point of failure to the system.

The most popular non-auto-incremental algorithm is **Universally unique identifier** - UUID. It’s a 128-bit value, which can be generated by several standard algorithms(RFC 4122) with a probability of duplicated values close to zero. A lot of projects use UUIDs as identifiers for their entities. I also heard that some projects use UUIDs as identifiers, but usual auto-incremental columns as a primary key.

There is a package for working with UUIDs in PHP - **ramsey/uuid**. It implements some algorithms of RFC 4122 standard. Now we can write our tests:

```
final class Client
{
    public static function register(
```

```
        UuidInterface $id,
        Email $email): Client
    {
        $client = new Client($id, $email);
        $client->record(new ClientRegistered($client->id));

        return $client;
    }
}

trait CreationTrait
{
    private function createUuid(): UuidInterface
    {
        return Uuid::uuid4();
    }

    private function createEmail(): Email
    {
        static $i = 0;

        return Email::create("test$i@test.test");
    }
}

class ClientTest extends UnitTestCase
{
    use CreationTrait;

    public function testRegister()
    {
        $client = Client::register($this->createUuid(),
            $this->createEmail());

        $this->assertEventsHas(ClientRegistered::class,
            $client->releaseEvents());
    }
}
```

I’ve implemented a simple assertion (**assertEventsHas**) in the base **UnitTestCase** class which checks that an event of the specified class exists in a given array. Do you remember black box and white box testing? Here is an example of black box testing. Test doesn’t get an email property with **getEmail()** getter or another way and compare it with the email that was used in creation. It just checks that the needed event was recorded.

Another test case:

```
class JobApplyTest extends UnitTestCase
{
    use CreationTrait;

    public function testApply()
    {
        $job = $this->createJob();
```

```
        $freelancer = $this->createFreelancer();

        $freelancer->apply($job, 'cover letter');

        $this->assertEventsHas(FreelancerAppliedForJob::class,
            $freelancer->releaseEvents());
    }

    public function testApplySameFreelancer()
    {
        $job = $this->createJob();
        $freelancer = $this->createFreelancer();

        $freelancer->apply($job, 'cover letter');

        $this->expectException(
            SameFreelancerProposalException::class);

        $freelancer->apply($job, 'another cover letter');
    }

    private function createJob(): Job
    {
        return Job::post(
            $this->createUuid(),
            $this->createClient(),
            JobDescription::create('Simple job', 'Do nothing'));
    }
}
```

These tests are just describing our core logic. Simple applying for a job. The case of same freelancer applying to job again. Tests check pure domain logic. There are no mocks, stubs, database working, etc. There is no trying to get the proposals array and checking something there. You can ask anyone who is not a developer but knows a domain and he will understand the logic of these tests. And these tests are definitely easy to write and they can be written together with logic.

Well, this domain is not complex enough to show the advantages of extracting it and building with convenient unit testing, but keep in mind the Monopoly game. Complex logic is much simpler to implement and support when it is covered by unit tests and extracted from any infrastructure (database, HTTP, etc.).

Creation of a good domain model is not a trivial task. I can recommend two books: Classical **“Domain-Driven Design: Tackling Complexity in the Heart of Software”** by **Eric Evans** and **“Implementing Domain-Driven Design”** by **Vaughn Vernon**. There are a lot of new concepts with examples from practice which can change the way you build a domain

model: **Aggregate root**, **Bounded context**, **Ubiquitous language**… After reading these great books, you will maybe understand that my current model isn't ideal, but it suits nicely for demo purposes. I am just trying to show the possibility of building a pure domain model in PHP.

Mapping model to database

Now, after building some logic, it's time to map our entities and value objects to database tables and fields. I'll use Doctrine library for that. laravel-doctrine/orm package provides a convenient integration of it to Laravel projects. Doctrine allows us to use different ways to configure mapping. Example with PHP array:

```
return [
    'App\Article' => [
        'type'   => 'entity',
        'table'  => 'articles',
        'id'     => [
            'id' => [
                'type'     => 'integer',
                'generator' => [
                    'strategy' => 'auto'
                ]
            ],
        ],
        'fields' => [
            'title' => [
                'type' => 'string'
            ]
        ]
    ]
];
```

Some developers prefer to leave entities without any information about database mapping and this outside configuration is a good option for them. But most developers just use annotations - tags added to phpDoc. Java has native support for annotations, PHP doesn't, but Doctrine analyzes them and uses them to make mapping much more convenient. Example with annotation mapping:

```
use Doctrine\ORM\Mapping AS ORM;

/** @ORM\Embeddable */
final class Email
{
    /**
     * @var string
     * @ORM\Column(type="string")
     */
```

```
    private $email;
    //...
}

/** @ORM\Embeddable */
final class Money
{
    /**
     * @var int
     * @ORM\Column(type="integer")
     */
    private $amount;
    // ...
}

/** @ORM\Entity */
final class Freelancer
{
    /**
     * @var Email
     * @ORM\Embedded(class = "Email", columnPrefix = false)
     */
    private $email;

    /**
     * @var Money
     * @ORM\Embedded(class = "Money")
     */
    private $hourRate;
}

/**
 * @ORM\Entity()
 */
final class Job
{
    /**
     * @var Client
     * @ORM\ManyToOne(targetEntity="Client")
     * @ORM\JoinColumn(nullable=false)
     */
    private $client;

    //...
}
```

I use **Embeddable** annotation for my value objects and **Embedded** to use them in other classes. Each annotation has parameters. Embedded requires the class parameter to know what to embed, and optional **columnPrefix** for the field name generation.

There are also annotations for different relation types: one to many, many to many, etc.

```
/** @ORM\Entity */
final class Freelancer
```

```
{
    /**
     * @var UuidInterface
     * @ORM\Id
     * @ORM\Column(type="uuid", unique=true)
     * @ORM\GeneratedValue(strategy="NONE")
     */
    private $id;
}

/** @ORM\Entity */
final class Proposal
{
    /**
     * @var int
     * @ORM\Id
     * @ORM\Column(type="integer", unique=true)
     * @ORM\GeneratedValue(strategy="AUTO")
     */
    private $id;
}
```

Each entity should have an **Id** field. UUID fields have type “uuid”, which means they will be stored in char(36) columns. UUID values have standard string representations. Example: e4eaaaf2-d142-11e1-b3e4-080027620cdd Alternatively they can be stored as 16-byte binary values (check **ramsey/uuid-doctrine** package).

Proposal entities are the part of **Job** unit and they will never be created outside. It doesn’t need an **id** value, but Doctrine asks to have it, so we can use an auto-incremental column and forget about it.

I don’t want to copy and paste Doctrine documentation to this book. There are some problems during mapping domain to database with Doctrine (for example, it still doesn’t support nullable embeddables), but in most cases it’s possible to implement it.

When Doctrine fetches an entity from database, it creates an object of the needed class without using a constructor and just fills needed fields with values without using any setter methods. It uses PHP Reflection magic for that. As a result: objects don’t feel the database. Their lifecycle is natural.

```
$freelancer = new Freelancer($id, $email);

$freelancer->apply($job, 'some letter');

$freelancer->changeEmail($anotherEmail);

$freelancer->apply($anotherJob, 'another letter');
```

There is a lot of database fetching and at least one persisting between each method call, which can be executed in different servers, but the object doesn't feel them. It just lives as it lives in some unit test case. Doctrine tries to make all needed infrastructure work to allow the objects to live naturally.

Migrations

It's time to create our database. Laravel migrations is a possible way to do it, but Doctrine suggests some magic here: doctrine migrations. After installing the **laravel-doctrine/migrations** package and running "**php artisan doctrine:migrations:diff**", fresh migration will appear in **database/migrations** folder:

```
class Version20190111125641 extends AbstractMigration
{
public function up(Schema $schema)
{
$this->abortIf($this->connection->getDatabasePlatform()->getName() != '\
sqlite',
    'Migration can only be executed safely on \'sqlite\'.');

$this->addSql('CREATE TABLE clients (id CHAR(36) NOT NULL --(DC2Type:uu\
id)
, email VARCHAR(255) NOT NULL, PRIMARY KEY(id))');
$this->addSql('CREATE TABLE freelancers (id CHAR(36) NOT NULL --(DC2Typ\
e:uuid)
, email VARCHAR(255) NOT NULL, hourRate_amount INTEGER NOT NULL,
PRIMARY KEY(id))');
$this->addSql('CREATE TABLE jobs (id CHAR(36) NOT NULL --(DC2Type:uuid)
, client_id CHAR(36) NOT NULL --(DC2Type:uuid)
, title VARCHAR(255) NOT NULL, description VARCHAR(255) NOT NULL,
PRIMARY KEY(id))');
$this->addSql('CREATE INDEX IDX_A8936DC519EB6921 ON jobs (client_id)');
$this->addSql('CREATE TABLE proposals (id INTEGER PRIMARY KEY
AUTOINCREMENT NOT NULL
, job_id CHAR(36) DEFAULT NULL --(DC2Type:uuid)
, freelancer_id CHAR(36) NOT NULL --(DC2Type:uuid)
, cover_letter VARCHAR(255) NOT NULL, hourRate_amount INTEGER NOT NULL)\
');
$this->addSql('CREATE INDEX IDX_A5BA3A8FBE04EA9 ON proposals (job_id)');
$this->addSql('CREATE INDEX IDX_A5BA3A8F8545BDF5 ON proposals (freelanc\
er_id)');
}

public function down(Schema $schema)
{
$this->abortIf($this->connection->getDatabasePlatform()->getName() != '\
sqlite',
    'Migration can only be executed safely on \'sqlite\'.');

$this->addSql('DROP TABLE clients');
$this->addSql('DROP TABLE freelancers');
```

```
$this->addSql('DROP TABLE jobs');
$this->addSql('DROP TABLE proposals');
}
}
```

I used sqlite for this test project. Well, they look ugly compared with clean Laravel migrations, but Doctrine can auto-generate them! “**doctrine:migrations:diff**” command analyzes current database and entities’ metadata and generates a migration, which should change the database structure for domain needs. For an empty database it will be the creation of all needed tables. For a case when some field is added to some entity, it will be just adding this field to the needed table.

I think that’s enough about Doctrine. It definitely allows a developer to build pure domain and effectively map it to the database. As I said before, after extracting Domain layer, Application layer only makes an orchestration between infrastructure and domain.

```
final class FreelancersService
{
    /** @var ObjectManager */
    private $objectManager;

    /** @var MultiDispatcher */
    private $dispatcher;

    public function __construct(
        ObjectManager $objectManager, MultiDispatcher $dispatcher)
    {
        $this->objectManager = $objectManager;
        $this->dispatcher = $dispatcher;
    }

    /**
     * Return freelancers's id.
     *
     * @param \App\Domain\ValueObjects\Email $email
     * @param \App\Domain\ValueObjects\Money $hourRate
     * @return UuidInterface
     */
    public function register(
        Email $email, Money $hourRate): UuidInterface
    {
        $freelancer = Freelancer::register(
            Uuid::uuid4(), $email, $hourRate);

        $this->objectManager->persist($freelancer);
        $this->objectManager->flush();

        $this->dispatcher->multiDispatch(
            $freelancer->releaseEvents());

        return $freelancer->getId();
```

```
    }
    //...
}
```

Here, UUID is generated in the Application layer. It can be generated a bit earlier. I heard about some projects which ask clients to generate UUIDs.

```
POST /api/freelancers/register
{
    "uuid": "e4eaaaf2-d142-11e1-b3e4-080027620cdd",
    "email": "some@email.com"
}
```

I think this way is canonical. Client asks the app to make an action and provides all needed data. Simple “200 OK” response is enough for the client. It already has an id and can continue to work.

Doctrine’s ObjectManager::persist method puts the entity to persisting queue. **ObjectManager::flush** method persists everything in persisting queue to database. Lets check a non-creation action:

```
interface StrictObjectManager extends ObjectManager
{
    /**
     * @param string $entityName
     * @param $id
     * @return null|object
     */
    public function findOrFail(string $entityName, $id);
}

final class DoctrineStrictObjectManager
    extends EntityManagerDecorator
    implements StrictObjectManager
{
    /**
     * @param string $entityName
     * @param $id
     * @return null|object
     */
    public function findOrFail(string $entityName, $id)
    {
        $entity = $this->wrapped->find($entityName, $id);

        if($entity === null)
        {
            throw new ServiceException(...);
        }

        return $entity;
    }
}
```

Here I extended the standard **ObjectManager** with **findOrFail** method, which does the same thing as the **findOrFail** method in Eloquent entities. The difference is only in the exception object. Eloquent generates an **EntityNotFound** exception, which transforms to a 404 http error. My **StrictObjectManager** will be used only in write operations and if some entity could not be found, it's more like a validation error (do you remember Laravel's "exists" validation rule?), not an error which should return a 404 response.

```
abstract class JsonRequest extends FormRequest
{
    public function authorize()
    {
        // Don't check authorization in request classes
        return true;
    }

    /**
     * Get data to be validated from the request.
     *
     * @return array
     */
    protected function validationData()
    {
        return $this->json()->all();
    }
}

final class JobApplyDto
{
    /** @var UuidInterface */
    private $jobId;

    /** @var UuidInterface */
    private $freelancerId;

    /** @var string */
    private $coverLetter;

    public function __construct(UuidInterface $jobId,
        UuidInterface $freelancerId, string $coverLetter)
    {
        $this->jobId = $jobId;
        $this->freelancerId = $freelancerId;
        $this->coverLetter = $coverLetter;
    }

    public function getJobId(): UuidInterface
    {
        return $this->jobId;
    }

    public function getFreelancerId(): UuidInterface
    {
        return $this->freelancerId;
    }
```

```
    public function getCoverLetter(): string
    {
        return $this->coverLetter;
    }
}

final class JobApplyRequest extends JsonRequest
{
    public function rules()
    {
        return [
            'jobId' => 'required|uuid',
            'freelancerId' => 'required|uuid',
            //'coverLetter' => optional
        ];
    }

    public function getDto(): JobApplyDto
    {
        return new JobApplyDto(
            Uuid::fromString($this['jobId']),
            Uuid::fromString($this['freelancerId']),
            $this->get('coverLetter', '')
        );
    }
}
```

JsonRequest is the base class for API requests, where data is provided as JSON in the request body. **JobApplyDto** is a simple DTO for a job apply action. **JobApplyRequest** is a **JsonRequest**, which makes needed validation and creates a **JobApplyDto** object.

```
final class FreelancersController extends Controller
{
    /** @var FreelancersService */
    private $service;

    public function __construct(FreelancersService $service)
    {
        $this->service = $service;
    }

    public function apply(JobApplyRequest $request)
    {
        $this->service->apply($request->getDto());

        return ['ok' => 1];
    }
}
```

Controllers become very simple. They just provide data from request objects to service action.

```php
final class FreelancersService
{
    public function apply(JobApplyDto $dto)
    {
        /** @var Freelancer $freelancer */
        $freelancer = $this->objectManager
            ->findOrFail(Freelancer::class, $dto->getFreelancerId());

        /** @var Job $job */
        $job = $this->objectManager
            ->findOrFail(Job::class, $dto->getJobId());

        $freelancer->apply($job, $dto->getCoverLetter());

        $this->dispatcher->multiDispatch(
                    $freelancer->releaseEvents());

        $this->objectManager->flush();
    }
}
```

Application layer action is also simple. It asks the object manager to fetch needed entities and asks them to make a business action. The main difference from Eloquent is the flush method call. Application layer doesn't have to try to save each entity to database. Doctrine remembers all entities that were fetched and determines all changes that were made there. Here, in flush method, it will find that a new object was added to the proposals property of job entity and insert this row to the database! This magic allows us to not think about how to store everything in the database. Domain entities just add new entities to its own properties, or just changes something. Doctrine persists all changes, even in deep relations. Application layer just calls **ObjectManager::flush** method.

Of course, everything has a price. Doctrine code is much more complicated than Eloquent's. While working with Eloquent, I can always go to its sources and understand why it behaves like that. I can't say the same about Doctrine :) Its configuration and working with complex select queries is harder than working with Eloquent.

You can check the full source of this super basic project in my Github:

https://github.com/adelf/freelance-example

I've added some CQRS pattern implementations there, so it is better to read the next chapter before.

Error handling in domain layer

In the “Error handling” chapter I suggested using an unchecked **BusinessException** to reduce **@throws** tags amount, but for complex domains it’s not a good idea. Exceptions can be thrown deep in the domain and at some level domain can react to them. Even in our simple example, **Freelancer** asks **Job** to add proposal, **Job** asks own proposals to check compatibility and an exception is thrown there.

Proposal object throws an exception (‘This freelancer already made a proposal’) but it doesn’t know a context. In the context of adding a new proposal, this exception goes to high levels (controller, global event handler) and is just shown to the user. In another context, caller wants to know what exactly was wrong. In a complex domain, it’s important to know what can happen in each method call, so using checked exceptions is a good option.

```
// BusinessException becomes "checked"
abstract class BusinessException
    extends \Exception {...}

final class SameFreelancerProposalException
    extends BusinessException
{
    public function __construct()
    {
        parent::__construct(
            'This freelancer already made a proposal');
    }
}

final class Proposal
{
    //...

    /**
     * @param Proposal $other
     * @throws SameFreelancerProposalException
     */
    public function checkCompatibility(Proposal $other)
    {
        if($this->freelancer->equals($other->freelancer)) {
            throw new SameFreelancerProposalException();
        }
    }
}

final class Job
{
    //...
```

```
    /**
     * @param Proposal $newProposal
     * @throws SameFreelancerProposalException
     */
    public function addProposal(Proposal $newProposal)
    {
        foreach($this->proposals as $proposal)
        {
            $proposal->checkCompatibility($newProposal);
        }

        $this->proposals[] = $newProposal;
    }
}
```

Another condition can be added with new requirements: make some kind of auction and don't allow user to add proposal with an hour rate higher than the existing one (not very smart requirement, but anyway). New exception will be added to **@throws** doc block. It will result in a cascade of changes in all callers. They also should add an additional **@throws** doc block.

Another problem: sometimes caller needs to know all possible problems. "same freelancer" and "too high hour rate" issues can happen at the same time, but only one exception will be thrown. Creating a new exception to store all the reasons why this proposal is not compatible with others and make the **checkCompatibility** method code much more complicated is not a good option.

That's why I often hear that developers prefer something like the **FunctionResult** objects from the "Error handling" chapter for their domain. There is no problem with returning an object with all issues from the checkCompatibility method. Caller's code becomes a bit more noisy. It's a trade-off again.

Conclusion

Extracting a Domain layer is a big step in project evolution. It's much more optimal to make it in the beginning, but the architect should estimate the complexity of domain. If a domain is just a simple CRUD with very little additional logic, there is no sense in extracting a Domain layer. It may cost a lot of time and money without significant benefits.

On the other hand, for complex domains it's definitely a correct choice. Writing pure domain logic with effortless unit testing is a big advantage. It is not easy to switch from anemic models (when Eloquent entities store only data and logic is implemented outside) to rich models (where data and logic are implemented in the same classes). It's the same as switching from procedural programming to object oriented. It requires some time and some practice, but benefits for some projects are also big.

As you see, getter methods (**getEmail**, **getHourRate**) are not needed to implement write operations (apply for a job) according to **Information Hiding** principle. If some getter of class **A** is used in class **B** logic, class **B** begins to know too much about class **A** inner structure. Each class **A** change might result in class **B** (and all other getter callers) changes. It easily becomes hard to support systems where using other class getters is a usual practice. Each change there becomes very dangerous. Popular developers' joke for these systems: "It works? Please, don't touch it!"

Sadly, systems also have a user interface where all this inner data should be shown. Getters have to be implemented because people want to see emails and hour rates. When entities have getters, it becomes so easy to violate **Information Hiding** and other good principles, so easy to make a mistake, especially in projects with many developers. Is there a way to not implement getters in our domain? Let's talk about it in the next chapter!

11. CQRS

Reading and writing - different responsibilities?

As we mentioned in the previous chapter, getters are not needed for a correct domain model. There are only entities and “write” methods, but the user interface has to show information to users. Is the getter method for entities the only solution, or there are some others? Let’s try to invent something!

Database stored procedures and views

The first project of my professional career was a huge application with logic in stored procedures and database views. There were thousands of them. I was writing a client for it in C++. Database view is just a saved select query which can be used later as a read-only table (some DBMS allow to write there, but it’s a bad practice).

```
CREATE TABLE t (qty INT, price INT);
INSERT INTO t VALUES(3, 50);
CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

View example from MySQL documentation contains a new ‘value’ field which doesn’t exist in the table.

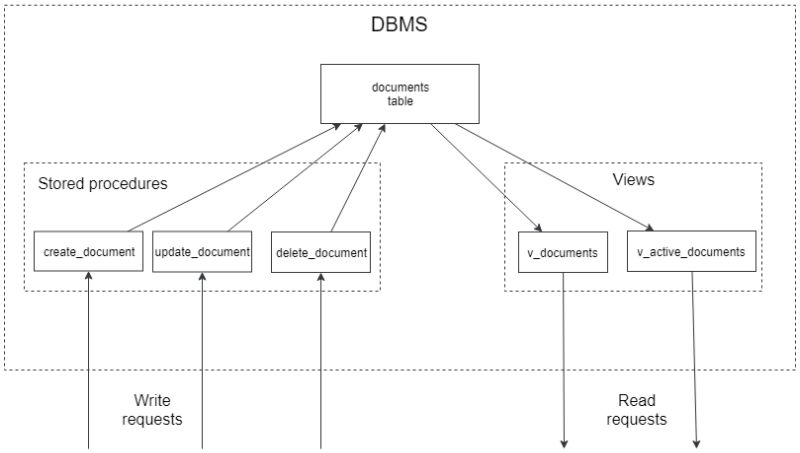
Stored procedure is just a set of instructions written as a special procedural extension of SQL(**PL/SQL** in Oracle, **Transact-SQL** for MSSQL and others). It’s like a PHP function, which is executed inside the database engine.

```
PROCEDURE raise_salary (
    emp_id NUMBER,
    amount NUMBER
  ) IS
```

```
BEGIN
  UPDATE employees
    SET salary = salary + amount
    WHERE employee_id = emp_id;
END;
```

Example of very simple procedure in PL/SQL.

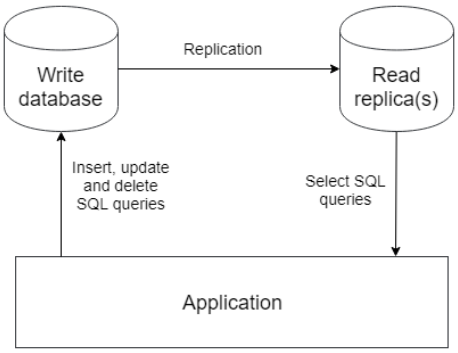
As I mentioned, the system was huge, there was a ton of logic. Without some restrictions, this kind of system easily becomes totally unsupportable and non-stable. There was some structure of views and stored procedures for each entity:



Tables were like the private fields of a class, inaccessible to select, update and other operations. While writing this book, I realized that these stored procedures and views implemented an Application layer for this system. I'm talking about this example because here it is very easy to see how different write and read operations are. They use absolutely different types of database objects - procedures and views.

Master-slave replication

Another case: databases in projects where loading becomes high. First thing developers can do to reduce loading to a database is to leave one database only for write operations and one or several databases for read operations. It's a "master-slave replication".

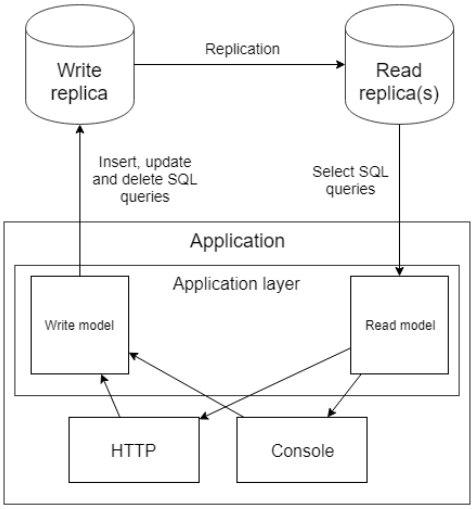


All changes go to the master (write) database and then replicate to slave (read) replicas. The same pattern: write requests to database go to one place, read requests to another.

Sometimes the replication process goes slow and read replicas contain old data. Users can change some data in the app but continue to see the old version of it. The same can happen when data caching was implemented not very carefully. An architecture of systems with one database and cache looks very similar to systems with write and read databases. Cache is a kind of read storage.

These slow replications and cached old values issues will be auto-fixed after some time. Read replicas will catch up current data. Cached values will expire. Anyway, if the user changed something, he will see the result, sometimes not immediately, but he will. This type of consistency is called “eventual”. Eventual consistency is a common attribute for systems with different stores for write and read data.

Developers should keep in mind eventual consistency by implementing Application service. All queries should use a write database connection during processing of a write operation. Not cached values, not fetched from read stores. This condition forces the segregation of Application layer code which works with write operations from code working with read operations.



But it’s not the only reason.

Typical Application layer class

```
final class PostService
{
    public function getById($id): Post{}
    public function getLatestPosts(): array{}
    public function getAuthorPosts($authorId): array{}

    public function create(PostCreateDto $dto){}
    public function publish($postId){}
    public function delete($postId){}
}
```

Set of write methods with set of read methods. Manipulations with this class might be not convenient.

Let’s try to implement data caching for read methods. If we implement caching inside **PostService** read methods, these methods will have at least two responsibilities: data fetching and caching. The most common solution is to use the **Decorator** pattern as I did in the “Dependency Injection” chapter. I have to extract an interface or abstract class from **PostService** class, but I have a problem with naming! interface **PostService**, but how to name the class, implementing it? Or maybe “class **PostService** implements **PostServiceInterface**”?

Another issue: decorator class for caching will cache read methods, but not touch write ones. This makes us realize that the **PostService** class has two responsibilities: write and read operations.

Let's leave write methods in **PostService** class and make something new for read:

```
final class PostService
{
    public function create(PostCreateDto $dto){}
    public function publish($postId){}
    public function delete($postId){}
}

interface PostQueries
{
    public function getById($id): Post;
    public function getLatestPosts(): array;
    public function getAuthorPosts($authorId): array;
}

final class DatabasePostQueries implements PostQueries{}

final class CachedPostQueries implements PostQueries
{
    /** @var PostQueries */
    private $baseQueries;

    /** @var Cache */
    private $cache;

    public function __construct(PostQueries $baseQueries,
        Cache $cache)
    {
        $this->baseQueries = $baseQueries;
        $this->cache = $cache;
    }

    public function getById($id): Post
    {
        return $this->cache->remember('post_' . $id,
            function() use($id) {
                return $this->baseQueries->getById($id);
            });
    }
    //...
}
```

Looks much better! Segregating write service class and read queries makes refactoring and other manipulations much easier.

Report queries

Report queries easily show the different natures of read and write models. Complex queries with ‘group by’, ‘union’ and aggregation functions… When developers try to use Eloquent entities for those kinds of queries, it always looks ugly. Entities built for write operations are not intended to be used as report data.

After some time, an obvious solution: to use **Structured Query Language**(SQL) comes to mind. Using raw SQL is much easier and natural for report queries. Data fetched from these queries are stored in simple classes (like DTO) or just in php arrays. This is a simple example of using a totally different model (service and “entity” classes) for the same data.

Laravel API resources

Laravel 5.5 version introduced API resources. They can be used to transform the data of the base Eloquent entity. It’s an attempt to solve the problem of different data needed in read queries and the several responsibilities of the Eloquent entity class. But they are not true read models, they still use an Eloquent entity to fetch the data. Creating fully independent read models will remove the read responsibility from the Eloquent entity.

Command Query Responsibility Segregation

Command Query Responsibility Segregation(CQRS) pattern suggests to have fully independent read and write models. Model here is a set of services working with database, Application layer, entities, value objects, etc.

Read and write models can be built with different ways and technologies. Write model with Doctrine or custom data mapper and read model with Active Record ORM or just raw SQL queries. Technologies and architecture for each model are chosen according project needs.

For the application from the previous chapter with a write model implemented using Doctrine, a read model can be implemented simply by Eloquent:

```php
namespace App\ReadModels;

use Illuminate\Database\Eloquent\Builder;
use Illuminate\Database\Eloquent\Model;

abstract class ReadModel extends Model
{
    public $incrementing = false;

    protected function performInsert(Builder $query)
    {
        throw new WriteOperationIsNotAllowedForReadModel();
    }

    protected function performUpdate(Builder $query)
    {
        throw new WriteOperationIsNotAllowedForReadModel();
    }

    protected function performDeleteOnModel()
    {
        throw new WriteOperationIsNotAllowedForReadModel();
    }

    public function truncate()
    {
        throw new WriteOperationIsNotAllowedForReadModel();
    }
}

final class WriteOperationIsNotAllowedForReadModel
    extends \RuntimeException
{
    public function __construct()
    {
        parent::__construct(
            "Write operation is not allowed for read model");
    }
}
```

A base class for Eloquent read models. It just overrides all write operations and blocks them.

```php
final class Client extends ReadModel{}

final class Freelancer extends ReadModel{}

final class Proposal extends ReadModel{}

final class Job extends ReadModel
{
    public function proposals()
    {
        return $this->hasMany(Proposal::class, 'job_id', 'id');
    }
}

final class ClientsController extends Controller
{
```

```
    public function get(UuidInterface $id)
    {
        return Client::findOrFail($id);
    }
}

final class FreelancersController extends Controller
{
    public function get(UuidInterface $id)
    {
        return Freelancer::findOrFail($id);
    }
}

final class JobsController extends Controller
{
    public function get(UuidInterface $id)
    {
        return Job::findOrFail($id);
    }

    public function getWithProposals(UuidInterface $id)
    {
        return Job::with('proposals')->findOrFail($id);
    }
}
```

Very basic implementation. Just entities called from controllers. As you see, a read model can be trivial, even for a complex write model with layered architecture. It can be refactored by introducing the ***Queries** or **Repository** classes, with caching decorators and other improvements. The big advantage is that a read model can make these improvements, but the write model won't be touched!

The case with a complex read and an average write domain is also possible. Once I had been participating in a high-load content project. Write model wasn't very complex, so we just extracted the Application layer and used Eloquent entities there. But the read model contained a lot of different complicated queries and the cache was actively used. So, for the read model we took simple read model entities, which were usual classes and contained only public fields, like DTO.

Conclusion

As each pattern, CQRS has advantages and disadvantages. It allows to build read and write parts of an application independently, which can help to reduce the complexity of the write (remove getters and other functionality from write entities) or read (use plain objects and raw SQL queries) part.

On the other hand, in most cases it’s a duplication of entities, some Application layer part, etc. It will take a lot more time to build two models instead of only one.

Read and write models often need some synchronization. Tasks for adding some new fields to entities contain at least two sub-tasks for read and write models. Application has to be covered well by functional tests.

12. Event sourcing

1. e4 - e5
2. Nf3 - Nc6
3. Bb5 - a6

Have you played chess? Even if not, you know this game. Two players just move the pieces. And it is the best example of pattern that I want to talk about.

The kings game

When chess lovers want to know about some chess grandmasters game, the final position on the board is not very interesting. They want to know about each move!

1. d4 - Nf6
2. Bg5

“What a joke from World champion!!!”

The meaning of the current position on the board fully depends on moves done before:

1. The most important: who is moving next? The position can be switched from winning to losing based on this.
2. Castling is possible only if the king and rook haven’t moved before.
3. En passant capture is possible only right after a target pawn double-step move.

Let’s create a chess playing web application. How we will store the games? I see two options:

1. store current position on the board, but with some additional information: who moves next, castling possibilities and some info about the last move for en passant capture. Moves can be stored in another “table”, just for history.
2. store all moves and get current position on the board by “replaying” all these moves each time.

There are two main notations for computer chess:

FEN - stores current position with all needed additional information Example:

```
rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2
```

PGN - stores all moves Example:

```
[Event "F/S Return Match"]
[Site "Belgrade, Serbia JUG"]
[Date "1992.11.04"]
[Round "29"]
[White "Fischer, Robert J."]
[Black "Spassky, Boris V."]
[Result "1/2-1/2"]

1. e4 e5 2. Nf3 Nc6 3. Bb5 a6
...
42. g4 Bd3 43. Re6 1/2-1/2
```

As you see, both ideas are okay, but which one to choose?

First way is traditional for web applications: we always store only the current state in database tables. Sometimes, we also have some history tables and store the logs there: how our data was changed.

Second way looks weird. Getting the current position by replaying each move is too slow, but having full information about the game might be very useful.

Let’s imagine two applications and analyze both options. First app stores chess games in table: id, current position, long/short castling possibility for whites/blacks, en passant field(if exists)

Second app just stores all moves for each game.

Application requirements change over time, here I purposely forgot some chess rules about draw. Fifty-move rule: the game can be declared a draw if no capture has been made and no pawn has been moved in the last fifty moves. This rule was added to remove the possibility of infinite games.

How our two systems will react to this change?

First system has to add a new field to table: amount of moves made without capturing and pawn movement(FEN chess notation also has this value). This new field value will be calculated after each move. The problem is with current playing games. The history of it is stored in some other table, but it doesn't participate in game logic.

Second system just implements this 50-move logic and it fluently will work, even for current playing games without changing anything on storage.

Looks like the second system is more prepared for changes. The second rule about draws: threefold repetition. A draw is declared if the same position occurs three times.

I don't know how the first app's developers will implement this change. It needs to have information about all previous positions. Second application developers again just implement this logic, without any changes in storage.

As you see, in chess domain is very important to have all historical information and not just as some logs, but as an active domain logic participant. But what if the second application becomes very popular and a lot of people watch chess games there? Calculating the current position for each watcher is not very effective due to performance issues. Obvious solution from previous chapter: CQRS. A write model system needs all moves, but a read model only needs a current position. After each move written to the write model, the system can calculate the current position on the board and write it to some table for read requests.

First system stored the current position on the board as a main source of information and all moves in the history table for some analytics needs. The second system, on the contrary, stores all moves as a main source and the

current position for a read model. As a result, the second system is more prepared for changes in core logic.

I chose chess because it is the best example of true Event Sourcing domain. Event Sourcing pattern suggests to store all changes in the system as a sequence of events.

Instead of having **posts** table:

```
PostId, Title, Text, Published, CreatedBy
```

All changes are stored in **post_events** table, which is append-only(only insert and select SQL queries is acceptable):

```
PostId, EventName, EventDate, EventData
```

Or in special event stores.

Events:

- **PostCreated**(Title, Text, CreatedBy)
- **PostPublished**(PublishedBy)

Well, blog posts are definitely not the right domain for Event Sourcing pattern. The logic there very rarely depends on the history of posts. Storing the whole application state as a sequence of events has these advantages:

- developers can “debug” each entity and understand how exactly it went to its current state.
- whole application state can be calculated for each moment of time.
- any state, based on historical data, can be calculated for each entity, even for old ones (if created_at and updated_at fields were forgotten, they can be added later and values will be calculated correctly for each entity).
- any logic, based on historical data, can be implemented and immediately start to work, even for old entities, not just for new ones. Example, some task tracker system: if some task was assigned to the same user 3 times - subscribe project manager to this task.

There are a lot of real industries where the current state is not the most important data:

- Balance of your bank account is just a cached value of all money transactions made on this account.
- Insurance calculations are fully based on events that happened before.
- Medical data is always historical.
- Accounting systems are working only with events.

Many industries are regulated and systems have to store audit logs, but if they are just secondary logs, their correctness is hard to prove. Having all “logs” as a primary source of data proves their correctness by default.

The real-life technical examples of ES are modern version control systems (like git) and blockchain technology.

Git stores all data as a sequence of changesets. Changeset contains events like: file A created with this content, some lines inserted to file B to this position, file C was deleted.

Blockchain is an append-only list of information blocks, which is linked using cryptography. Each block contains a cryptographic hash of the previous block, a timestamp, and transaction data (cryptocurrency transactions or document changes).

Database management systems store all operations that modify data (insert, update, delete SQL queries) in a special transaction log and sometimes use it as a main source of data. Replication process between master and slave databases are usually the transfering of a transaction log and the replaying of all operations there in slave instance.

Write model unit tests

Let’s take a look again to unit tests for a model written in the Domain layer chapter.

```
class JobApplyTest extends UnitTestCase
{
    public function testApplySameFreelancer()
    {
```

```
        $job = $this->createJob();
        $freelancer = $this->createFreelancer();

        $freelancer->apply($job, 'cover letter');

        $this->expectException(SameFreelancerProposalException::class);

        $freelancer->apply($job, 'another cover letter');
    }
}
```

Instead of manually creating an object in the needed state and testing the behavior in this state, tests have to create this entity in its initial state and run some commands to get the entity to its needed state. This unit test repeats the ES model.

Events are the only information these models push outside and unit tests can only check these events. What if some entity will record a successful event but forget to change its state? If the entity is well covered by tests, some other test will fail. If **Job** entity won't add proposal, **SameFreelancer** test will fail. In a complex entity with an enormous amount of possible states, some states might not be tested (do you remember chess and Monopoly?). Let's check a very simple case with publishing posts.

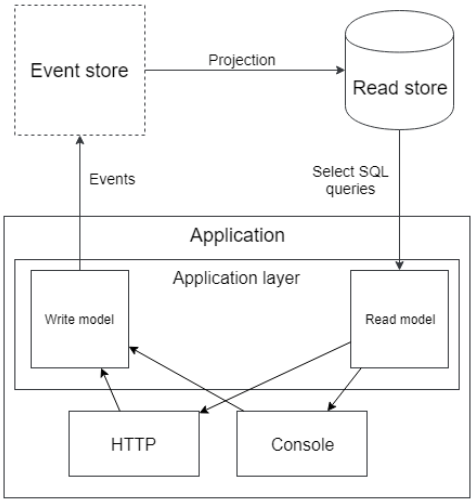
```
class Post
{
    public function publish()
    {
        if (empty($this->body)) {
            throw new CantPublishException();
        }

        //$this->published = true;

        $this->record(new PostPublished($this->id));
    }
}

class PublishPostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulPublish()
    {
        // initialize
        $post = new Post('title', 'body');

        // run
        $post->publish();

        // check
        $this->assertEventsHas(
            PostPublished::class, $post->releaseEvents());
    }
}
```

Test is okay, but published field isn’t updated in database. Our read model won’t get the proper value and only functional tests will catch this problem (if they were written for this case). In ES systems, the event itself is a main source, the data which will be written to the database (or special store). So, unit tests become much more natural in ES systems. They definitely check the whole behavior.

World without magic

How are entities from the Domain layer chapter stored in a database? Entity classes there are very closed, all fields are private, and only action methods are public. First, Doctrine analyzes the php-files with entity class sources, fetches meta information about entities, fields, and how to map them to the database (it stores this meta-information cached then). After **persist** and **flush** commands, it uses the full power of dark PHP reflection magic to get all needed values and stores them to database. What will happen if all magic disappears? In a world without magic, we can’t build domains like in the Domain layer chapter.

We can forget about **Information Hiding** and implement getters just to allow a data mapper to save the entities’ states to a database. Or just implement Event Sourcing! Events are public. They can be stored in the database instead of the current entities’ state.



It looks very similar to the schema of usual CQRS application. Write model stores events instead of data needed in the read model. All data needed for the read model is only the projection of these events:

- Traditional current state of entities in the tables
- Full-text search indices (SphinxSearch or Elasticsearch)
- Special statistics data for reports (sometimes stored in special aggregate tables)
- Other data for reading

Events are the true source of all data the system has generated. Using them as a write model is very natural, especially if the system is already using a CQRS pattern.

Implementing ES

Today (in the beginning of 2019) the best PHP library to implement an ES pattern is **prooph**. It also has a perfect demo project, which I want to show here - **prooph/proophessor-do**. One of the core concepts of Domain driven design, **Ubiquitous language**, is very well presented there by widely used value objects and named constructors.

Each root entity has the VO for id:

```
interface ValueObject
{
    public function sameValueAs(ValueObject $object): bool;
}


final class TodoId implements ValueObject
{
    /** @var UuidInterface */
    private $uuid;

    public static function generate(): TodoId
    {
        return new self(Uuid::uuid4());
    }

    public static function fromString(string $todoId): TodoId
    {
        return new self(Uuid::fromString($todoId));
    }

    private function __construct(UuidInterface $uuid)
    {
```

```
        $this->uuid = $uuid;
    }

    public function toString(): string
    {
        return $this->uuid->toString();
    }

    public function sameValueAs(ValueObject $other): bool
    {
        return \get_class($this) === \get_class($other)
            && $this->uuid->equals($other->uuid);
    }
}
```

TodoId value object just represents UUID value. There is also a **ValueObject** interface to compare value objects with each other.

```
final class TodoWasPosted extends AggregateChanged
{
    /** @var UserId */
    private $assigneeId;

    /** @var TodoId */
    private $todoId;

    /** @var TodoStatus */
    private $todoStatus;

    public static function byUser(UserId $assigneeId, TodoText $text,
        TodoId $todoId, TodoStatus $todoStatus): TodoWasPosted
    {
        /** @var self $event */
        $event = self::occur(...);

        $event->todoId = $todoId;
        $event->assigneeId = $assigneeId;
        $event->todoStatus = $todoStatus;

        return $event;
    }

    public function todoId(): TodoId {...}

    public function assigneeId(): UserId {...}

    public function text(): TodoText {...}

    public function todoStatus(): TodoStatus {...}
}
```

An event which creates a todo item. **AggregateChanged** is the base class for all ES-events in **Prooph**. The named constructor is used to make event creation code look as a natural language sentence:

TodoWasPosted::byUser(…). I've hidden some technical stuff, which is not important right now.

Each entity has to extend **AggregateRoot** class. The main parts of it:

```
abstract class AggregateRoot
{
    /**
     * List of events that are not committed to the EventStore
     *
     * @var AggregateChanged[]
     */
    protected $recordedEvents = [];

    /**
     * Get pending events and reset stack
     *
     * @return AggregateChanged[]
     */
    protected function popRecordedEvents(): array
    {
        $pendingEvents = $this->recordedEvents;

        $this->recordedEvents = [];

        return $pendingEvents;
    }

    /**
     * Record an aggregate changed event
     */
    protected function recordThat(AggregateChanged $event): void
    {
        $this->version += 1;

        $this->recordedEvents[] =
                        $event->withVersion($this->version);

        $this->apply($event);
    }

    abstract protected function aggregateId(): string;

    /**
     * Apply given event
     */
    abstract protected function apply(AggregateChanged $event);
}
```

Here is the same pattern for storing events inside the entity as we used in the Domain layer chapter. The important difference is the **apply** method. Entities can change their state only by applying the event to itself. As you remember, entities' current state restores by replaying all events from the

beginning, so entities only write the events with **recordThat** method and it redirects them to apply method, which does real changes.

```
final class Todo extends AggregateRoot
{
    /** @var TodoId */
    private $todoId;

    /** @var UserId */
    private $assigneeId;

    /** @var TodoText */
    private $text;

    /** @var TodoStatus */
    private $status;

    public static function post(
        TodoText $text, UserId $assigneeId, TodoId $todoId): Todo
    {
        $self = new self();
        $self->recordThat(TodoWasPosted::byUser(
            $assigneeId, $text, $todoId, TodoStatus::OPEN()));

        return $self;
    }

    /**
     * @throws Exception\TodoNotOpen
     */
    public function markAsDone(): void
    {
        $status = TodoStatus::DONE();

        if (! $this->status->is(TodoStatus::OPEN())) {
            throw Exception\TodoNotOpen::triedStatus($status, $this);
        }

        $this->recordThat(TodoWasMarkedAsDone::fromStatus(
          $this->todoId, $this->status, $status, $this->assigneeId));
    }

    protected function aggregateId(): string
    {
        return $this->todoId->toString();
    }

    /**
     * Apply given event
     */
    protected function apply(AggregateChanged $event): void
    {
        switch (get_class($event)) {
            case TodoWasPosted::class:
                $this->todoId = $event->todoId();
                $this->assigneeId = $event->assigneeId();
                $this->text = $event->text();
                $this->status = $event->todoStatus();
                break;
            case TodoWasMarkedAsDone::class:
```

```
                $this->status = $event->newStatus();
                break;
            }
        }
    }
```

Little part of the **Todo** entity. The difference between this entity and entities from the Domain layer chapter is small, but important: entities' state fully depends only on events.

When **Prooph** has been asked to store entity, it just gets its id and all events to store by calling popRecordedEvents method and saves them to event store (it might be a database table or something else). To get an entity in its current state by id, it finds all events with this id, creates an empty object of needed class and replays all events one by one with the apply method.

```
final class Todo extends AggregateRoot
{
    /** @var null|TodoDeadline */
    private $deadline;

    /**
     * @throws Exception\InvalidDeadline
     * @throws Exception\TodoNotOpen
     */
    public function addDeadline(
        UserId $userId, TodoDeadline $deadline)
    {
        if (! $this->assigneeId()->sameValueAs($userId)) {
            throw Exception\InvalidDeadline::userIsNotAssignee(
                $userId, $this->assigneeId());
        }

        if ($deadline->isInThePast()) {
            throw Exception\InvalidDeadline::deadlineInThePast(
                $deadline);
        }

        if ($this->status->is(TodoStatus::DONE())) {
            throw Exception\TodoNotOpen::triedToAddDeadline(
                $deadline, $this->status);
        }

        $this->recordThat(DeadlineWasAddedToTodo::byUserToDate(
            $this->todoId, $this->assigneeId, $deadline));

        if ($this->isMarkedAsExpired()) {
            $this->unmarkAsExpired();
        }
    }
}
```

Another part of **Todo** entity: adding a deadline to a todo item. The first **if** statement makes an authorization job and it's a little violation of the Single Responsibility Principle, but for a demo project it's absolutely okay. Just try to read the **addDeadline** methods code. It looks like a normal text in natural English, thanks to well-named constructors and value objects. The deadline object is not just a **DateTime**, it's a **TodoDeadline** object, with all needed methods, like **isInThePast**, which make a client's code very clean.

I don't want to go deeper in the model of this demo project. I highly recommend to review it for everyone who wants to learn how to build an ES application - **https://github.com/prooph/proophessor-do**

Projections are special objects which transform events from write model to data for a read model. Almost all ES systems have obvious projections building traditional tables with the current state of entities.

```
final class Table
{
    const TODO = 'read_todo';
    //...
}


final class TodoReadModel extends AbstractReadModel
{
    /**
     * @var Connection
     */
    private $connection;

    public function __construct(Connection $connection)
    {
        $this->connection = $connection;
    }

    public function init(): void
    {
        $tableName = Table::TODO;

        $sql = <<<EOT
CREATE TABLE `$tableName` (
`id` varchar(36) COLLATE utf8_unicode_ci NOT NULL,
`assignee_id` varchar(36) COLLATE utf8_unicode_ci NOT NULL,
`text` longtext COLLATE utf8_unicode_ci NOT NULL,
`status` varchar(7) COLLATE utf8_unicode_ci NOT NULL,
`deadline` varchar(30) COLLATE utf8_unicode_ci DEFAULT NULL,
`reminder` varchar(30) COLLATE utf8_unicode_ci DEFAULT NULL,
PRIMARY KEY (`id`),
KEY `idx_a_status` (`assignee_id`,`status`),
KEY `idx_status` (`status`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8 COLLATE=utf8_unicode_ci;
EOT;
```

```
        $statement = $this->connection->prepare($sql);
        $statement->execute();
    }

    public function isInitialized(): bool
    {
        $tableName = Table::TODO;

        $sql = "SHOW TABLES LIKE '$tableName';";

        $statement = $this->connection->prepare($sql);
        $statement->execute();

        $result = $statement->fetch();

        if (false === $result) {
            return false;
        }

        return true;
    }

    public function reset(): void
    {
        $tableName = Table::TODO;

        $sql = "TRUNCATE TABLE `$tableName`;";

        $statement = $this->connection->prepare($sql);
        $statement->execute();
    }

    public function delete(): void
    {
        $tableName = Table::TODO;

        $sql = "DROP TABLE `$tableName`;";

        $statement = $this->connection->prepare($sql);
        $statement->execute();
    }

    protected function insert(array $data): void
    {
        $this->connection->insert(Table::TODO, $data);
    }

    protected function update(
        array $data, array $identifier): void
    {
        $this->connection->update(
            Table::TODO,
            $data,
            $identifier
        );
    }
}
```

This class represents a table for storing todo items. **init**, **reset** and **delete** methods are used when the system needs to create or rebuild the projection. In this class, they just create and drop/truncate a **read_todo** table. **insert** and **update** methods just do insert/update SQL queries.

The same class can be created for building/updating a full-text search index, statistics data, or just logging all events to file (this is not the best way to use projection, because all events are already stored in event store).

```
$readModel = new TodoReadModel(
    $container->get('doctrine.connection.default'));

$projection = $projectionManager
    ->createReadModelProjection('todo', $readModel);

$projection
->fromStream('todo_stream')
->when([
    TodoWasPosted::class
        => function ($state, TodoWasPosted $event) {
        $this->readModel()->stack('insert', [
            'id' => $event->todoId()->toString(),
            'assignee_id' => $event->assigneeId()->toString(),
            'text' => $event->text()->toString(),
            'status' => $event->todoStatus()->toString(),
        ]);
    },
    TodoWasMarkedAsDone::class
        => function ($state, TodoWasMarkedAsDone $event) {
        $this->readModel()->stack(
            'update',
            [
                'status' => $event->newStatus()->toString(),
            ],
            [
                'id' => $event->todoId()->toString(),
            ]
        );
    },
    // ...
])
->run();
```

This is a configuration of projection. It uses a **TodoReadModel** class and transforms the events to commands for this class. **TodoWasPosted** event will create a new row in this table. **TodoWasMarkedAsDone** event will change a status field for the specified **id**. After running all events from the beginning, **read_todo** table will contain the current state of all todo items in the system. Usual data change workflow for ES system is: fetching entity (aggregate root) from the event store, calling the command (**markAsDone**

or **addDeadline**), getting all events generated by this command (it may contain more than 1 event), storing them to the event store, calling all projections, some of them immediately (usually the traditional current state table projection), some of them by queue.

Unique data in ES systems

One of the disadvantages of Event Sourcing systems is that data in the event store is impossible to check by constraints like unique indexes. In traditional DBMS, unique index for the users.email field makes sure that the system will never have users with the same email. Entities in ES systems are fully independent. Different user entities with the same email but different id values can live together without any issues, but system requirements don't allow this.

Some systems just use a table from the read model to check uniqueness (unique index there helps), but in case of race condition, this also can allow one to write 2 users with the same email to the event store (projection obviously will fail trying to insert the second record). Some systems just create a special table with one unique field and insert the value there, before storing an event.

Conclusion

Event Sourcing is a very powerful pattern. It allows to easily implement logic based on historical data. It can help to prove that your "audit log" is correct (for regulated industries). It also helps systems to be more prepared for changes, especially if the changes are based on historical data.

Disadvantages are also very big. ES is much more expensive than traditional ways. For most applications, ES is too heavy and not optimal when compared with the extracted Domain layer or even "everything in controller" "pattern" which is usually called MVC. It also requires some skill level from team members. "Thinking by events" is very different from "thinking by database rows" and each new team member will spend a lot of time adapting to a project's style of thinking. Analyzing demo projects, like **proophessor-do**, or creating one's own simple ES projects, especially with non-standard domains, can help to understand the difficulties of

implementing the ES pattern and to decide whether to use ES in new projects.

13. Sagas

Multi-system “transactions”

I want to make a little chapter about another interesting use of events. I already wrote about database transactions, which allow us to have consistent data. They execute all queries inside, or roll them back. What if our system is big and contains several subsystems with their own databases? Each subsystem can be written using a different framework or even programming language. How can one make transactions between these subsystems? The Saga pattern solves this problem.

Saga is a sequence of local transactions and events, generated after each of them.

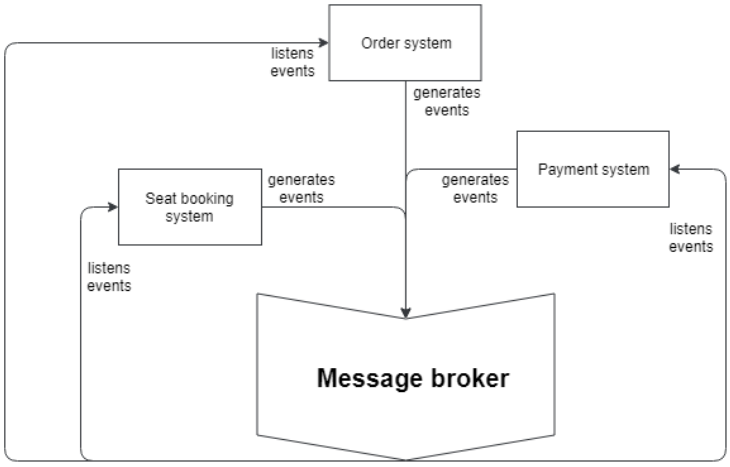
Let’s imagine some example. Air tickets selling system. Subsystems are:

- Order system with user interface (usually they are different systems)
- Aircraft seats booking
- Payment processing

Successful Saga:

1. Order system creates a new order.
2. Seat booking system reserves the seats in aircraft.
3. Order system asks user to enter payment details (card, for example).
4. Payment system processes the payment.
5. Seat booking system marks reserved seats as booked.
6. Order system marks the order as successful.

Systems have to talk with each other. Events are a good choice for that. They can be pushed to message queue systems (like RabbitMQ or Apache Kafka) and processed by subsystems listeners there.



Successful Saga with events:

1. Order system creates a new order (status=pending) in its own database and generates **OrderCreated** event.
2. Seat booking system catches this event and reserves the seats in aircraft. **SeatsReserved** event.
3. Order system catches this event and asks user to enter payment details (card, for example). **PaymentDetailsSet** event.
4. Payment system processes the payment. **PaymentProcessed** event.
5. Order system marks the order as payed. **OrderPayed** event.
6. Seat booking system marks reserved seats as booked. **SeatsBooked** event.
7. Order system marks the order as successful.

Each event contains the **OrderId** value to identify the order in each system.

Each step might be failed. Seats might be reserved by someone else. User can cancel the order at any step. Payment might be processed wrong. If Saga was failed in some step, all previous actions should be reverted. Reserved or booked seats should be freed. Money should be returned.

It should be implemented by compensating actions and ***Failed** events. If payment was failed, **PaymentFailed** event will be fired. Seat booking

system reacts to this event and frees the seats. Order system marks this order as failed.

UserCancelled event can be fired during each step by user. All systems will make compensation actions.

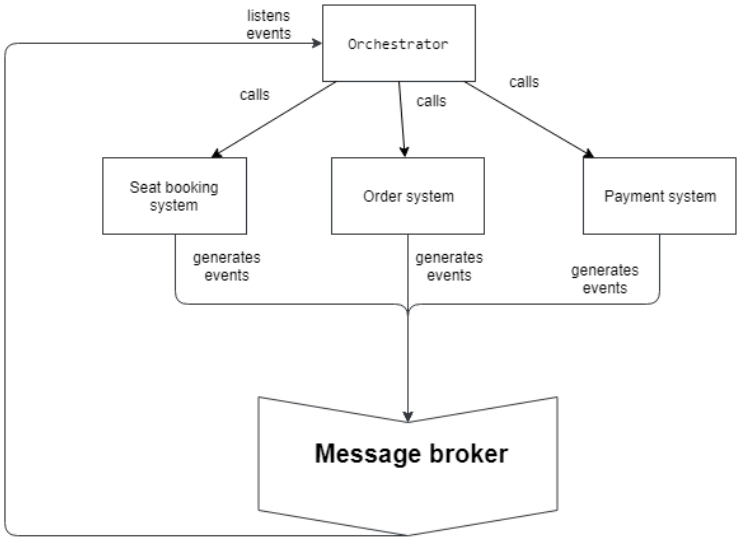
Saga is a high-level transaction. Each ***Failed** event causes a rollback action and the system returns to its initial state.

Orchestration

I see at least two problems in this Saga. Systems know too much about each other. Seat booking system should react to **PaymentFailed** event of payment system. And both them should react to **UserCancelled** UI event. It doesn't look like a big problem in this simple example, but real Sagas might have hundreds of possible events. Each change there can result in changes in many other systems. Can you imagine how difficult it might be to find all usages of some event? "Find usages" IDE feature doesn't work well in multi-system applications.

Another problem is that some ***Failed** events shouldn't rollback the whole Saga. Order system can ask user to choose another payment method on **PaymentFailed** event. It will be like a "partial rollback". Implementing this will increase the complexity of the Saga and will touch each system.

This Saga coordination way with pure events is called "Choreography-based Saga". Another way to coordinate Sagas is "Orchestration-based Saga". An additional object - orchestrator - makes all coordination between systems and they don't know about each other. Orchestrator might be an additional system or part of one of the Saga participants (in the order system, for example).



Orchestrator catches **OrderCreated** event and asks seat booking system to reserve the seats. After **SeatsReserved** event, it asks Order system to make needed actions and so on. Systems in Orchestration-based Sagas are fully independent and implement only its own responsibilities, leaving “event-reaction” functionality to orchestrator.

Sagas can solve a large number of issues of implementing business transactions by multiple systems, but they are, as each multi-system solution, very hard to support. Orchestrator reduces the dependencies between systems, which helps a lot, but the whole system will always be complex. Teams use visualisations like block diagrams to describe each Saga to have a full understanding how it works, but even diagrams become too complicated for some Sagas. Some projects are just too big and will always be complex.

14. Useful books and links

As I said in the beginning of this book, most patterns and techniques were observed superficially. This chapter contains some books and links which can help you to understand some of them better.

Classic

- **“Clean Code”** by Robert Martin
- **“Refactoring: Improving the Design of Existing Code”** by Martin Fowler

DDD

- **“Domain-Driven Design: Tackling Complexity in the Heart of Software”** by Eric Evans
- **“Implementing Domain-Driven Design”** by Vaughn Vernon

ES and CQRS

- https://aka.ms/cqrs - CQRS journey by Microsoft
- https://github.com/prooph/proophessor-do - Prooph library example project

Unit testing

- F.I.R.S.T Principles of Unit Testing