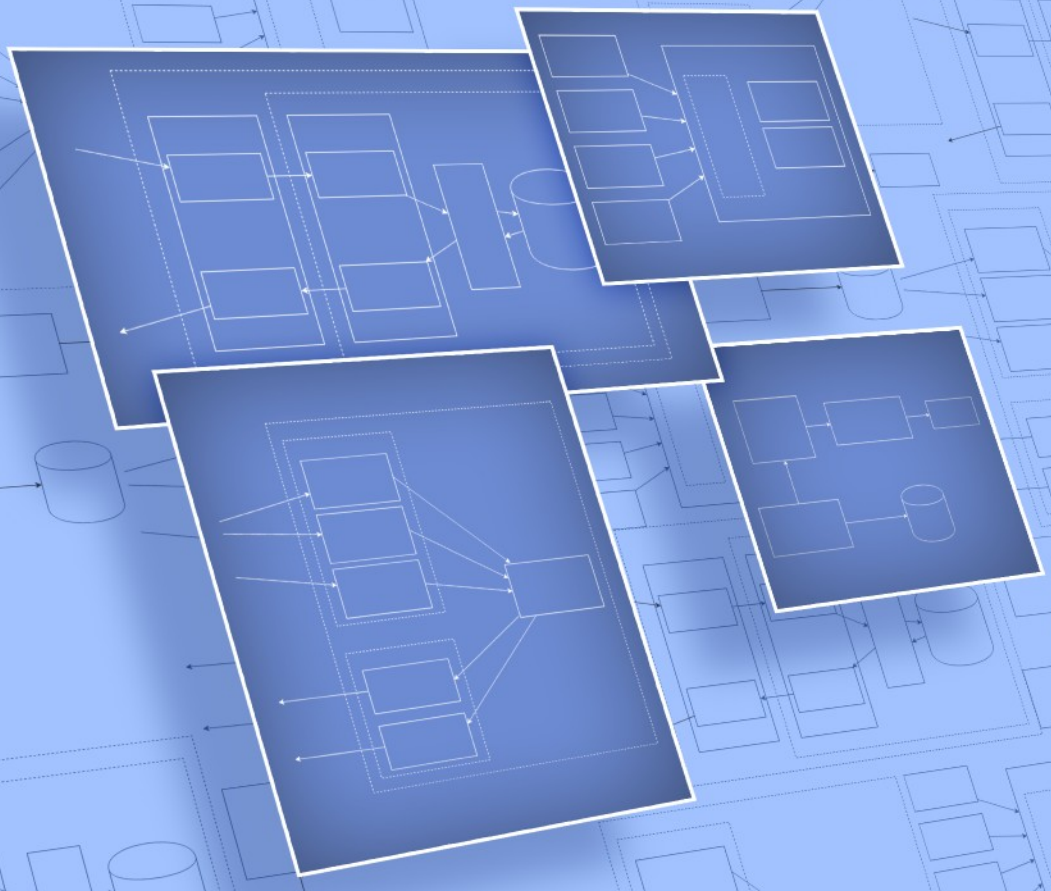
Adel F.

# 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

# 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:

```php
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');
}
```

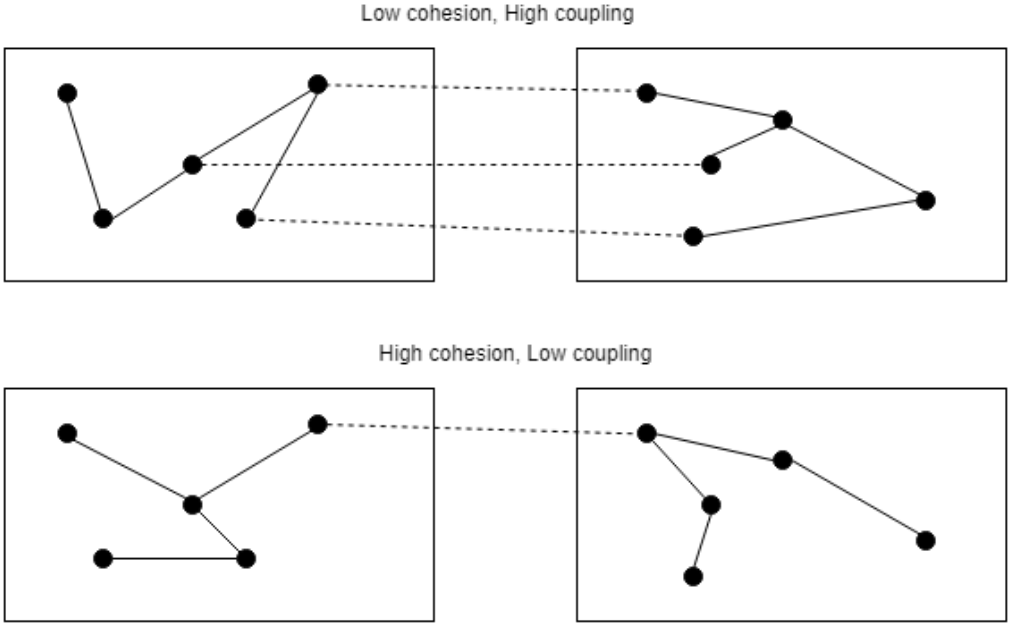
What kind of changes can this app have? Add/remove fields, add/remove entities… It's hard to imagine something more. I don't think this code violates SRP. Almost. Theoretically, redirecting to another route change is possible, but it's not very important. I don't see any reason to refactor this code. 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:

```
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

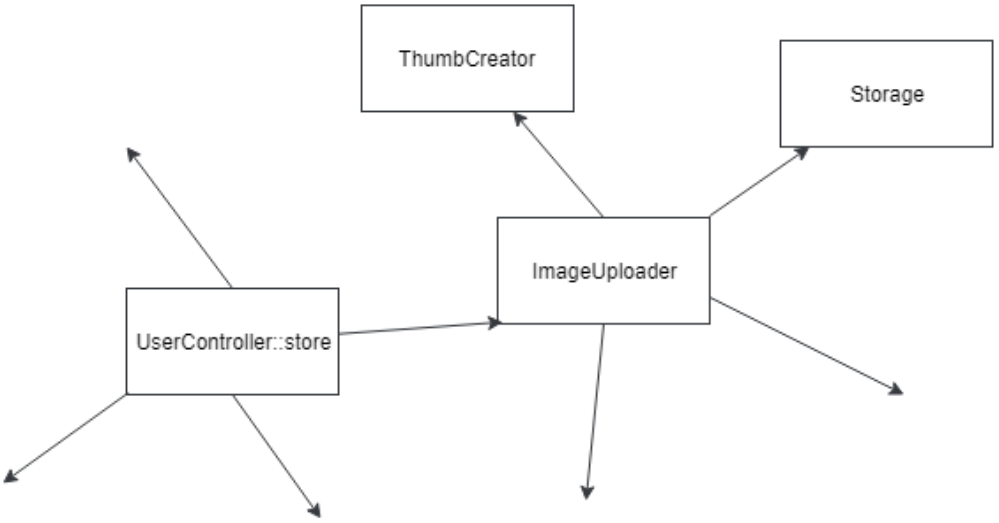
Well, we created **ImageUploader** class, but how do we use it in **UserController**::**store** method?

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

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

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

```php
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):

```php
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**:

```php
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

```php
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.

```
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.

```
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":

```
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;

    /** @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:

```php
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.

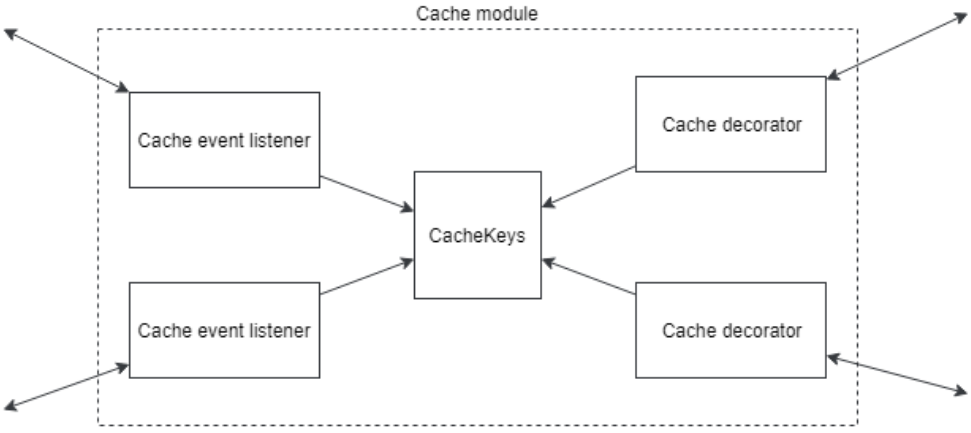
# Static methods

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

```php
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.

This is a free sample of the book. Full version is here - https://adelf.tech/2019/architecture-of-complex-web-applications[1]

[1]https://adelf.tech/2019/architecture-of-complex-web-applications