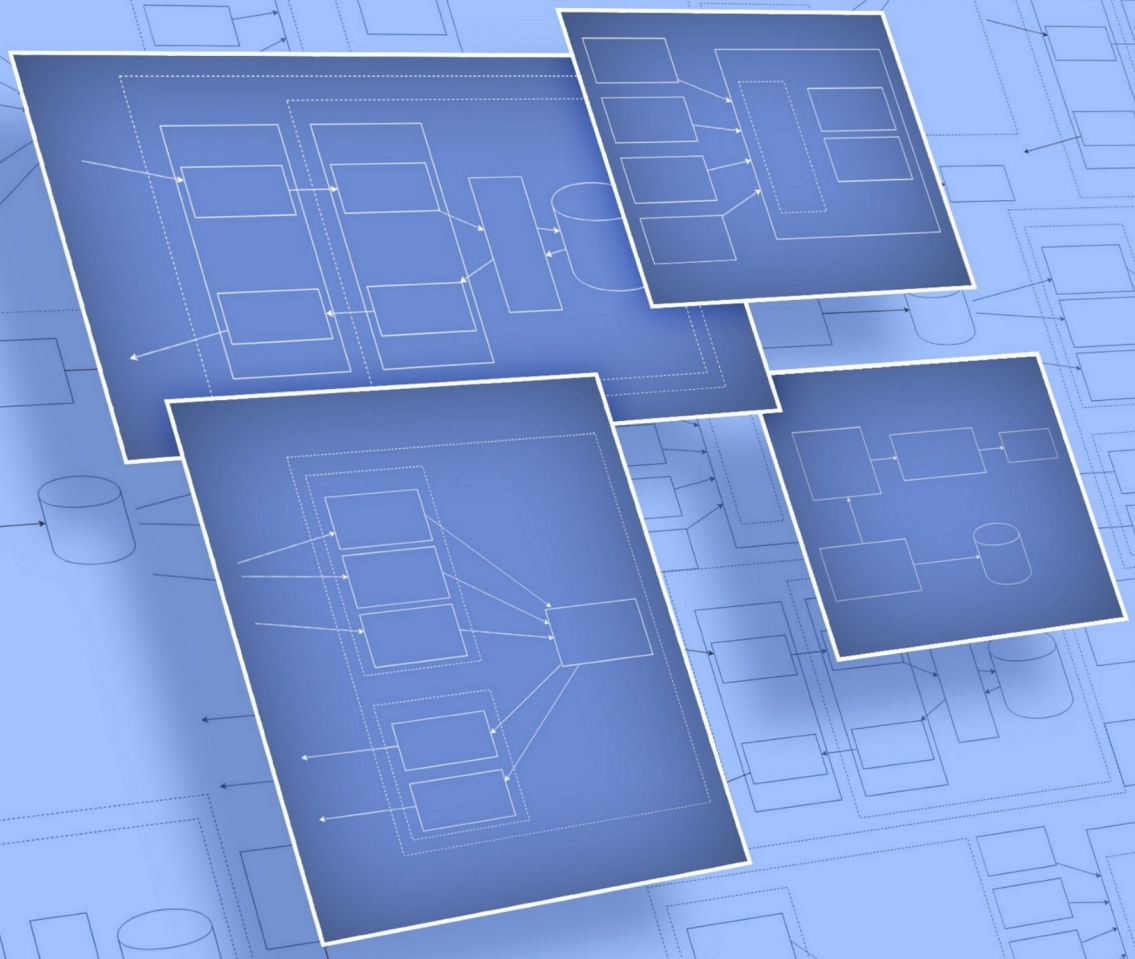
Официальный русский перевод

Adel F.

Architecture of complex web applications

With examples in Laravel(PHP)



Архитектура сложных веб приложений

Адель Файзрахманов

Оглавление

1. Предисловие **1**

2. Плохие привычки **4**
Проблемы роста 4
Выделение логики 7
Соблазнительная “простота” REST 11
Поклонение темной магии PHP 15
«Быстрая» разработка приложений (RAD) 17
Преждевременная оптимизация 21
Экономия строк кода 22
Прочие источники боли 24

3. Внедрение зависимостей **25**
Принцип Единственной Ответственности 25
Dependency Injection 30
Наследование 40
Пример с загрузкой картинок 46
Расширение интерфейсов 61
Трейты 67
Статические методы 77
Пара слов в конце главы 78

4. Безболезненный рефакторинг **80**
“Статическая” типизация 80
Шаблоны 84
Поля моделей 85
Laravel Idea 89

5. Слой Приложения 91
Передача данных запроса 94
Работа с базой данных 100
Сервисные классы или классы команд? 105
Пара слов в конце главы 107

6. Обработка ошибок 108
Исключения (Exceptions) 113
Базовый класс исключения 117
Глобальный обработчик 119
Проверяемые и непроверяемые исключения 121
Пара слов в конце главы 128

7. Валидация 130
Валидация связанная с базой данных 130
Два уровня валидации 134
Валидация аннотациями 136
Value objects 139
Объект-значение как композиция других значений 142
Объекты-значения и валидация 144
Пара слов в конце главы 147

8. События 148
Database transactions 149
Очереди 152
События 154
Использование событий Eloquent 156
Сущности как поля классов-событий 158
Пара слов в конце главы 161

9. Unit-тестирование 162
Первые шаги 162
Тестирование классов с состоянием 168
Тестирование классов с зависимостями 171
Типы тестов ПО 178
Тестирование в Laravel 179
Unit-тестирование Слоя Приложения 188

Стратегия тестирования приложения 195

10. Доменный слой 197
Когда и зачем? 197
Реализация Доменного слоя 205
Обработка ошибок в доменном слое 232
Пара слов в конце главы 234

11. CQRS 236
Чтение и запись - это разные ответственности? 236
Типичный сервисный класс 241
Command Query Responsibility Segregation 243
Пара слов в конце главы 247

12. Event sourcing 248
Игра королей 248
Unit-тестирование сущностей 254
Мир без магии 256
Реализация ES 258
Уникальные данные в ES-системах 270
Пара слов в конце главы 271

13. Заключение 272
Классика 272
DDD 272
ES и CQRS 272
Unit тестирование 273

1. Предисловие

> «Разработка ПО — это Искусство Компромисса», wiki.c2.com

Я видел множество проектов, выросших из простой «MVC» структуры. Часто разработчики объясняют шаблон MVC так: «View (представление) — это HTML-шаблоны, Model (модель) — это класс Active Record (например, Eloquent) и один контроллер, чтобы править всеми!». Хорошо, не один, но обычно вся дополнительная логика реализуется в классах-контроллерах. Контроллеры часто содержат огромное количество кода, реализующего разную логику (загрузку картинок, вызовы внешних API, работу с базой, и т.д.). Иногда некоторая логика выносится в «базовые» контроллеры, чтобы уменьшить количество дублированного кода, и они распухают тысячами строк. Одни и те же проблемы возникают как в средних проектах, так и в огромных порталах с миллионами посетителей в день.

Шаблон MVC был изобретен в 1970-х годах для графического интерфейса пользователя (GUI). Простые CRUD (Create, Read, Update и Delete) веб-приложения, в сущности, являются просто интерфейсом к базе данных, поэтому пере-изобретённый шаблон «MVC для веб» стал очень популярен. Однако, веб-приложения очень быстро перестают быть только лишь интерфейсом к базе данных. Что говорит шаблон MVC про работу с файлами (изображения, музыка, видео), внешними API, кэшэм? Что если у сущностей поведение отличается от простого Создать-Изменить-Удалить? Ответ простой: Модель в терминах MVC — это не только класс Active Record. Она содержит всю логику работы со всеми данными приложения. Больше 90% кода современного сложного веб-приложения — это Модель, и это не только ORM сущности, но и огромный пласт других классов. Создатель фреймворка Symfony

Fabien Potencier как-то написал: «Я не люблю MVC, потому что это не то, как работает веб. Symfony2 — это HTTP фреймворк; это Request/Response фреймворк» Я могу сказать то же самое про Laravel и многие другие фреймворки. Задача веб-приложения проста: получить запрос и отдать ответ. Чем сложнее приложение, тем выгоднее разработчикам понимать это и не думать о проекте в терминах MVC. Классы web-контроллеров, как и классы консольных команд, например, являются лишь точками входа в наше приложение. Описывать остальную часть как Model и View для многих приложений будет некорректным и только мешает пониманию.

Фреймворки такие, как Laravel, содержат кучу RAD-возможностей (Rapid Application Development - быстрая разработка приложений), которые позволяют разрабатывать приложения эффективно срезая некоторые углы. Они весьма полезны на стадии приложения «интерфейс для работы с базой данных», но часто становятся источником боли по мере развития. Я делал много рефакторингов просто, чтобы избавить приложения от таких возможностей. Вся эта автомагия и «удобные» валидации в стиле «быстро сходить в базу данных и проверить нет ли такого email в таблице» хороши, но разработчик должен полностью понимать как они работают и когда лучше от них отказаться.

С другой стороны, советы от крутых разработчиков в стиле «ваш код на 100% должен быть покрыт юнит-тестами», «не используйте статические методы» и «зависеть нужно только от абстракций» быстро становятся своеобразными карго-культами для некоторых проектов. Слепое следование им приводит к огромным потерям во времени. Я видел интерфейс **IUser** с более чем 50 свойствами (полями) и класс **User: IUser** со всеми этими свойствами скопированными туда (это был C# проект). Я видел огромное количество абстракций просто для того, чтобы достичь требуемого процента покрытия юнит-тестами. Некоторые эти советы могут быть неверно истолкованы, некоторые применимы только в конкретной ситуации, некоторые имеют

важные исключения. Разработчик должен понимать какую проблему решает шаблон, в каких условиях он применим, и самое важное, в каких условиях лучше от него отказаться.

Проекты бывают разные. К некоторым хорошо придутся определённые шаблоны и практики. Для других они будут излишни. Один умный человек сказал: «Разработка ПО — это всегда компромисс между краткосрочной и долгосрочной продуктивностью». Если мне нужен один функционал в другом месте проекта, то я могу просто скопировать его туда. Это будет очень продуктивно, но доставит проблемы в будущем. Почти каждое решение про рефакторинг или применение какого-либо шаблона представляет собой ту же дилемму. Иногда, решение не применять шаблон, который сделает код «лучше» будет более правильным, поскольку полезный эффект от него будет меньше, чем время затраченное на его реализацию. Балансирование между шаблонами, практиками, техниками, технологиями и выбор наиболее подходящей комбинации для конкретного проекта является наиболее важным умением разработчика/архитектора.

В этой книге я покажу наиболее частые проблемы, возникающие в процессе роста проекта и как разработчики обычно решают их. Причины данных решений весьма важная часть книги.

Я должен предупредить:

- Эта книга не для начинающих. Чтобы понимать описываемые проблемы вы должны поучаствовать хотя бы в одном проекте. В одиночку или в команде.
- Эта книга не пособие. Много шаблонов будут описаны поверхностно, с целью просто познакомить читателя с ними. Несколько полезных ссылок вас ожидает в конце книги.
- Примеры этой книги никогда не будут идеальными. Я могу назвать какой-то код «корректным» и найти кучу ошибок в нем в следующей главе.

2. Плохие привычки

> “Сегодня курение спасет жизни!”

Проблемы роста

В этой главе я попытаюсь показать, как обычно проекты растут и решают возникающие проблемы. Начнём с простого примера:

```
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)
{
    // все примерно так же, как и в store()
}
```

Пример практически скопирован из документации, и он показывает всю мощь и элегантность Laravel. Это хорошо, фреймворк и должен быть таким, предоставляющим хорошие инструменты для решения стандартных задач. Проблемы возникают тогда, когда нужно добавлять фичи, не укладывающиеся в такую элегантную структуру. Их начинают лепить одна к другой, не обращая внимания на архитектуру. Со стороны это выглядит как маленький симпатичный домик, к которому пристраивают одну часть за другой, пока это не станет выглядеть как страшная мешанина.

Причина в том, что разработчик, реализуя непростые требования, хочет оставаться в тепличных условиях маленького домика, тогда, когда для этой реализации явно нужно что-то побольше. Давайте посмотрим как это может происходить на практике.

Для нашего приложения появляются новые требования — добавить загрузку аватара и отправку email пользователю после создания.

```
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');
```

```
}
```

Какая-то логика должна быть скопирована в **update** метод, но, например, отправка email должна быть только после создания. Код всё ещё выглядит неплохо, но количество дублированного кода растет. Еще 3-4 таких добавления и у нас будут красоваться два уже весьма больших и почти одинаковых метода **store** и **update**. Тут часто совершается ошибка - в попытке устранить дублирование кода создается метод-монстр с названием, допустим, **updateOrCreateUser**. Этому методу нужен параметр, чтобы понимать, что он сейчас делает: “update” или “create”.
Сразу после выделения код обычно выглядит весьма прилично, но с добавлением новых требований, особенно разных для создания и изменения сущности, разработчик начинает осознавать, что попал в капкан. В одном из проектов я видел 700-строковый метод с большим количеством `if($update)`:

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

Стоит ли говорить, что в нем частенько заводились баги, которые было трудно отлавливать в такой каше. Ошибка проста - разработчик вынес **разную** логику, которая показалась ему похожей, в один метод. Выносить в отдельный метод или класс надо всегда только одинаковую логику. Логика создания или редактирования сущности может показаться одинаковой поначалу, но почти наверняка окажется разной по мере развития проекта.

Здесь я хочу отвлечься и поговорить про naming - процесс наименования элементов языка (переменных, методов и классов). Если стараться именовать методы по смыслу, то многое можно понять уже по имени метода. **updateOrCreateUser** - тут проблема видна

без какого-либо анализа. **Or**(**Или**) это явный знак как минимум двух разных действий - двух разных логик. С развитием проекта все такие логики имеют свойство всё сильнее и сильнее отличаться друг от друга, и находиться в одном месте им совершенно противопоказано.

Надо всегда стараться называть методы, классы и переменные по смыслу. Иногда даже просто попытка назвать их хорошо может навести на хорошие мысли, которые помогут улучшить дизайн вашего кода.

В случае почти одинаковых действий **create** и **update**, более правильным будет оставить каждую в своем методе и уже внутри каждого искать одинаковые действия, такие как загрузки изображений, выделять их в методы или классы с точными понятными именами. Результатом будут два метода с говорящими именами, с читабельным кодом, а бонусом будут выделенные методы и классы, которые, вполне вероятно, можно будет использовать в других местах приложения.

Выделение логики

На проект приходит новое требование — автоматически проверять загружаемые картинки на неподобающий контент. Некоторые разработчики просто добавят этот код в **store** метод и скопируют его в **update** метод. Более опытные выделят эту логику в новый метод контроллера и вызовут этот метод в обоих местах. Еще более опытные Laravel-разработчики найдут, что код для загрузки изображения стал довольно большим и создадут отдельный класс, например **ImageUploader**, который будет содержать логику загрузки изображений и проверки их содержимого на неподобающий контент.

```
class ImageUploader
{
    /**
     * @returns bool|string
     */
    public static function upload(UploadedFile $file) {...}
}
```

ImageUploader::upload метод возвращает **false** если загрузка не была успешной, например, при ошибке облачного хранилища или неприемлемом контенте. При удачной загрузке будет возвращен URL-адрес картинки.

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

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

Методы контроллера стали проще, поскольку логика загрузки картинок-аватарок вынесена в другой класс. Отлично! Если на проекте возникнет необходимость загружать другие картинки, то нужный класс уже готов к использованию. Необходимо только добавить новый параметр в метод **upload** — например, папку, куда сохранять картинки.

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

Новое требование — немедленно забанить пользователя, который загрузил неприемлемый контент. Звучит немного странно, учитывая неидеальную точность современных анализаторов изображений, но это было настоящим требованием на одном из моих проектов!

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

Новое требование — не банить пользователя, если неприемлемый контент был загружен в приватные места.

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

Когда я говорю «новое требование» это не означает, что оно появляется на следующий день. В больших проектах между этими «новыми требованиями» могут пройти месяцы или годы. Их реализацией могут заниматься другие разработчики, которые не понимают почему этот код был написан таким образом. Их задача — просто реализовать это требование в коде, по возможности сделав это быстро. Даже если им не нравится какая-то часть кода, им трудно оценить время на рефакторинг. А также, что более важно, трудно не сломать что-либо. Это довольно частая проблема.

Новое требование — приватные места пользователя должны иметь менее строгие правила проверки контента.

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

Последнее требование для этого примера — приложение не должно банить сразу. Только после нескольких попыток загрузить неприемлемый контент.

```
public static 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());
        }
    }
    //...
}
```

Этот код уже не так хорош. Функция загрузки изображения имеет кучу странных параметров про проверку контента и бан юзеров. Если процесс бана юзера изменится, разработчик должен открыть класс **ImageUploader** и реализовывать изменения там, что выглядит не очень логично. Читать код вызова метода **upload** все сложнее и сложнее:

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

В новых версиях PHP можно указывать именованные параметры и код может выглядеть так:

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

Это намного читабельнее, но является лишь попыткой скрыть настоящую проблему. Любой boolean-параметр для функции означает, что у нее внутри спрятано как минимум две логики, а каждый дополнительный boolean-параметр увеличивает это число, в некоторых случаях даже экспоненциально.

Такие параметры почти всегда означают нарушение Принципа единственной ответственности (Single Responsibility Principle). Класс **ImageUploader** теперь занимается далеко не только загрузкой изображений, но и кучей разных других дел. У него присутствуют и другие проблемы, но мы поговорим о них позже.

Соблазнительная “простота” REST

Подход RESTful очень популярен. Laravel-разработчики используют ресурсные контроллеры с готовыми методами store, update, delete, и т.д. даже для web роутов, не только для API. Он выглядит очень просто. Всего 4 глагола: **GET** (прочитать), **POST** (создать), **PUT/PATCH** (изменить) и **DELETE** (удалить).

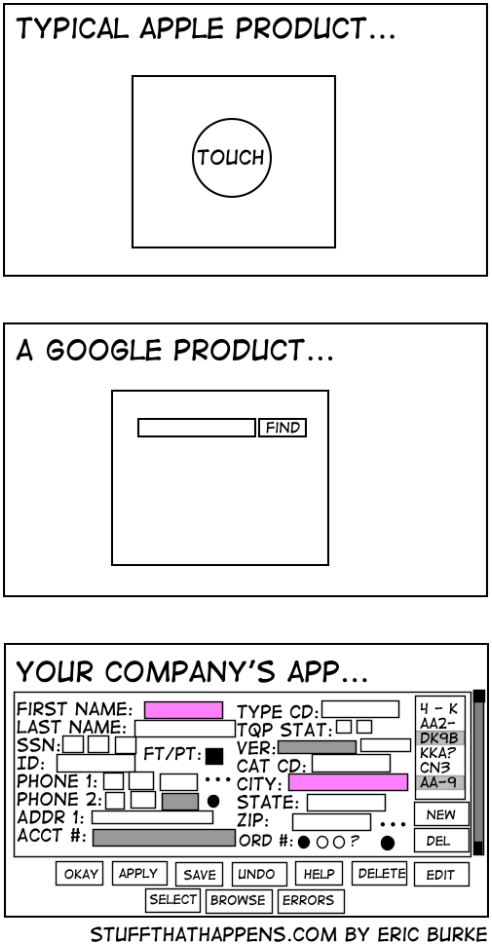
Он действительно весьма хорошо ложится на проекты, которые представляют собой те же самые простые операции над сущностями — обычные CRUD-приложения (Create, Read, Update, Delete) с формами для создания/редактирования и списками сущностей с кнопкой «Удалить». Но когда приложение становится более сложным, подход RESTful моментально становится весьма неуютным. Например, я погуглил фразу «REST API ban user» и первые три результата с примерами из документаций к разным API отличались разительно.

```
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

Там также была огромная таблица: какие статусы могут быть
изменены на какие и что при этом произойдет
```

Как видите, любой нестандартный глагол — и RESTful подход становится весьма неоднозначным, особенно для начинающих. Обычно все методы реализовываются через метод изменения сущности. Когда я спросил на одном из своих семинаров, «Как бы вы реализовали бан юзера в своем REST API?», первый ответ был:

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

Я часто вспоминаю картинку «типичный продукт Apple, типичный продукт Google» как лучшую иллюстрацию проблемы. Проблема эта в том, что разработчики, осознавая, что в конечном итоге изменения приведут к простому UPDATE SQL-запросу к таблице, соответствующей данной модели, начинают все эти изменения реализовывать через `update()` метод этой модели. На картинке мы видим результат крайней степени этой привычки, оказавший существенное влияние даже на интерфейс пользователя.



Нет ничего проще, чем написать `$model->update($request->all())`, но последствий такой простоты очень много и все они плохи. Дальше в этой книге мы придем к паре из них, здесь же я хочу остановиться на потере контроля. Когда код находится под контролем в каждой его точке понятно, что здесь происходит - какая логика здесь исполняется. Код `$model->update($request->all())` говорит нам о том, что мы просто обновляем строчку в базе данных данными HTTP-запроса.

Для приложения не предоставляющего абсолютно никакой дополнительной логики, кроме как интерфейса к строчкам базы данных - это совершенно нормально, ибо такова поставленная задача. Но сущности, даже простейшие, имеют привычку обрастать каким-то смыслом, вырастая из простого набора полей таблицы в некий объект с поведением. Пользователь - это не просто набор полей из таблички users. Это объект, модель чего-то настоящего из реального мира, который можно забанить или разбанить. Даже публикацию в блоге можно опубликовать или снять с публикации.

`$model->update($request->all())` сразу становится не настолько явным действием. Произошла ли здесь публикация поста? Забанили пользователя или нет? Мы не знаем! Контроль над кодом потерян, а он нужен: как только сущность обрастает поведением, рано или поздно мы захотим реагировать на это поведение. Посылкой email или очисткой кеша. Можно придумать много попыток восстановить этот контроль, но разделяются они на две категории: лечение причины и лечение последствий.

Вот пример лечения последствий, часто виденный мною в исходниках:

```
function afterUserUpdate(User $user)
{
    if (!$user->getOriginal('isBanned') && $user->isBanned) {
        // Отправить письмо о 'бане'
    }
}
```

Этот код кричит о проблеме. Данные каким-то волшебным образом изменились (скорее всего через `$user->update($data)`), а здесь мы пытаемся осознать что же реально произошло. Старайтесь изо всех сил избегать таких ситуаций. Нужно лечить причину потери контроля. Если нужно забанить пользователя нужен явный вызов команды ban: будь это метод `UserController::ban`, `$user->ban()`,

или класс `BanUserCommand`. А внутри всем всегда будет понятно, что реально происходит с сущностью. Это важно.

Поклонение темной магии PHP

Иногда разработчики не видят (или не хотят видеть) простого пути реализации чего-либо. Они пишут код с рефлексией, магическими методами или другими динамическими фичами языка PHP. Код, который было трудно писать и будет намного труднее читать. Я частенько этим грешил. Как каждый разработчик, я думаю.

Я покажу один веселый пример. Я написал простой класс для работы с ключами кэширования для одного из проектов. Ключи кэширования нужны как минимум в двух местах: при чтении из кэша и при удалении значений оттуда до срока. Очевидное решение:

```
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);
```

Помните догму «Не используйте статические функции!»? Почти всегда она верна, но это хороший пример исключения. Мы поговорим об этом в главе про внедрение зависимостей. Когда в другом проекте возникла такая же необходимость я показал этот класс разработчику сказав, что можно сделать также. Чуть погодя он сказал, что этот класс «не очень красивый» и показал свой вариант:

```
/**
 * @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('<огромный 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);
```

Этот код трансформирует строки вида «getUserById» в строки «USER_BY_ID» и использует значение константы с таким именем. Большое количество разработчиков, особенно те, кто помоложе, обожают писать подобный «красивый» код. Иногда этот код позволяет сэкономить несколько строк кода. Иногда нет. Но он всегда будет крайне сложным в отладке и поддержке. Разработчик должен подумать раз 10 прежде, чем использовать подобные «крутые» возможности языка.

«Быстрая» разработка приложений (RAD)

Некоторые разработчики фреймворков тоже любят динамические возможности и тоже реализуют подобную «магию». Она помогает быстро реализовывать простые мелкие проекты, но используя подобную магию разработчик теряет контроль над выполнением кода приложения и когда проект растет, это превращается в проблему. В прошлом примере было упущено использо-

вание констант `*::VERSION`, поскольку используя такую динамику, трудно как-либо изменить логику.

Другой пример: Laravel приложения часто содержат много подобного кода:

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

Laravel предлагает использовать «implicit route binding». Этот код работает так же, как и предыдущий:

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

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

Это действительно выглядит приятнее и позволяет избавиться от некоторого количества дублированного кода. Но здесь мы опять потеряли контроль над кодом. Смотря в данный код мы не знаем как именно сущность User была запрошена. В большинстве

случаев это не добавит никаких проблем и даже более того: если конкретно данному методу неважно откуда была взята сущность, например он просто покажет пользователю email данного юзера - это даже хорошо. Но бывают и другие ситуации.

Спустя некоторое количество времени, когда проект немного вырастет, разработчики начнут внедрять кеширование. Проблема в том, что кеширование можно применять для запросов чтения (GET), но не для запросов записи (POST). Подробнее об этом в главе про CQRS. Разделение операций чтения и записи станет намного больше, если проект начнет использовать разные базы данных для чтения и записи (это случается довольно часто в проектах с высокой нагрузкой). Laravel позволяет довольно легко сконфигурировать подобную работу с базами для чтения и записи. Продолжая использовать фичи фреймворка, мы будем вынуждены перейти с «implicit route binding» на «explicit route binding» и реализовать это как-то так:

```
Route::bind('user', function ($id) {
    // получить и вернуть закешированного юзера или 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');
```

Этот код выглядит очень странно и легко позволяет сделать ошибку. Это произошло потому, что тут опять лечат последствия, а не причину. Вместо явного запроса сущности по id разработчики использовали неявную «оптимизацию» и потеряли контроль над своим кодом, а такие попытки его вернуть лишь ухудшают ситуацию.

Замечаете один и тот же шаблон? Разработчик принимает

неудачное решение, которое кажется удачным поначалу. По мере усложнения проекта это решение все больше и больше усложняет жизнь. Вместо того чтобы собрать волю в кулак и признать, что решение было плохим, разработчик не исправляет его, а исправляет лишь его последствия, что делает код все хуже и хуже. Одним из главных умений разработчика является то шестое чувство, когда начинаешь понимать, что твой код “сопротивляется” изменениям - не хочет меняться легко и плавно, а требует все больше и больше усилий, заплаток и сделок с совестью для реализации таких изменений. Почувствовав такое сопротивление, надо искать его причины, найти те самые неудачные решения в дизайне кода, и исправить их. Это сделает проект намного более податливым к следующим изменениям. Именно это шестое чувство и позволяет большим проектам долго оставаться на плаву, не превращаясь в то самое “легаси”, которого мы все стараемся избежать при приеме на работу.

Фреймворки предлагают много возможностей потерять контроль над своим кодом. Нужно быть весьма осторожными с ними. Как небольшой подытог могу сказать следующее: чем меньше «магии» в коде, тем легче его читать и поддерживать. В очень редких случаях, таких как реализация библиотеки ORM, нормально использовать магию, но только там и даже тогда не стоит сильно увлекаться.

Изначальный код мог быть сокращен без использования route binding:

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

Нет смысла «оптимизировать» эту одну строчку кода. В дальнейшем при реализации кеширования можно будет явно отделить выборки с кешированием от выборок без него.

Преждевременная оптимизация

Тут разработчику может прийти мысль сразу подготовить код к будущим изменениям, например, заранее определить места, в которых нужно будет кешировать данные и выделить их в отдельные классы/методы. А то и реализовать кеширование сразу, несмотря на то, что проект в этом необходимости не испытывает. Это довольно тонкий момент. Разработчик делает ставку на то, что проект будет испытывать определенные проблемы и пытается заранее подготовить код к ним. Проблемой может быть неготовность к нагрузкам или неготовность к некоторым изменениям в коде (основная тема данной книги). Ставкой является то дополнительное время, которое понадобится разработчику или всей команде на реализацию этой подготовки. Если проект действительно в будущем испытает эти проблемы, то ставка будет оправдана, поскольку времени на исправление кода наверняка понадобилось бы больше.

Проблема в том, что предугадать проблемы удается весьма редко. Да, опытные разработчики из компаний FAANG или близким к ним, разрабатывая следующий сервис, наверняка знают в каких местах будут проблемы с нагрузкой и действительно могут довольно точно предугадывать такие моменты и экономить время, делая оптимизацию заранее. В большинстве же случаев попытки заранее оптимизировать приложение - это просто трата драгоценного времени.

Не стоит применять архитектурные решения из последних глав этой книги для простеньких приложений - это будут большие затраты с отрицательной пользой. Не стоит реализовывать кеширование или другие архитектурные решения для борьбы с

нагрузками для проекта, в котором нет уверенности хотя бы на 90 процентов в реальности и типе будущих нагрузок. Это тоже будет кучей времени выброшенной в мусорку, плюс почти наверняка добавит багов - не зря инвалидация кеша названа одной из главных проблем программирования. Если коротко - не занимайтесь преждевременной оптимизацией.

Экономия строк кода

Когда я учился программированию в университете, мы любили показывать друг другу примеры супер короткого кода. Обычно это была одна строка кода, реализующая какой-то алгоритм. Если автор этой строки пытался понять её спустя несколько дней — это могло занять несколько минут, но это все равно был «крутой» код.

В промышленной разработке крутость кода ценится намного меньше читабельности. Код должен быть понятен любому другому программисту при наименьших когнитивных затратах. Самый короткий код далеко не всегда самый читабельный. Не надо пытаться экономить байты, теряя чистоту и ясность кода. Обычно, несколько простых классов намного лучше, чем один, но сложный класс, и это работает со всеми элементами языка(методы, модули и т.д.).

Еще один пример с одного из моих проектов:

```
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()
   ]);
}
```

Разработчик хотел сэкономить пару строк кода и реализовал блокировку и разблокировку пользователя одним методом. Проблемы начались с наименования этого метода. Неудачное существительное **blockage** вместо естественных глаголов **block** и **unblock**. Каждый раз когда вам трудно как-либо назвать программный объект (класс, метод) - в нем таится какая-то проблема! Кроме очевидного нарушения логики - странного двойственного действия в одном методе, здесь также есть и проблема с конкурентностью. Когда два модератора одновременно захотят заблокировать одного и того же пользователя, первый его заблокирует, а второй — разблокирует.

Отдельные методы для блокировки и разблокировки - намного более логичное решение. Больше строк, но гораздо меньше неявности и других проблем.

Прочие источники боли

Я забыл рассказать о главном враге — Copy-Paste Driven Development. Просто скопировать логику в то место, где она понадобилась - очень продуктивно в краткосрочной перспективе. Не нужно заботиться о выделении этой логики в отдельный класс или метод и подключать ее в каждом месте. Но Вселенная ответит очень быстро и больно - эффективно поддерживать одну и ту же логику, размноженную в нескольких местах не удастся никому.

С дублированной логикой есть и обратная проблема. Иногда, хоть и довольно редко, логика бывает одинаковая в каждой строчке, но она разная по смыслу. Здесь соблазн выделить эти логики в один метод или класс еще более велик, чем с create и update. В будущем эти части кода легко могут оказаться разными из-за изменившихся требований, но разработчики часто попадают в ловушку, пытаясь устранить эту “дублированность” заранее. Совет очень простой - дублированная логика - это та логика, которая всегда будет одинаково меняться с изменением требований. Но более подробно мы будем об этом говорить в следующей главе.

3. Внедрение зависимостей

> “Дайте мне точку опоры и я переверну землю”

Принцип Единственной Ответственности

Вы, возможно, слышали о Принципе Единственной Ответственности (Single Responsibility Principle, SRP). Одно из его определений: «Каждый модуль, класс или функция должны иметь ответственность над единой частью функционала». Много разработчиков упрощают его до «класс должен делать что-то одно», но это определение не самое удачное. “Что-то одно” может быть одной строкой кода, или целой большой огромной задачей. Выделить границы здесь весьма непросто.

Роберт Мартин предложил другое определение, где заменил слово «ответственность» на «причину для изменения»: «Класс должен иметь лишь одну причину для изменения». «Причина для изменения» более удобный термин и мы можем начать рассуждать об архитектуре используя его. Почти все шаблоны и практики имеют своей целью лучше подготовить приложение к изменениям, но приложения бывают разные, с разными требованиями и разными возможными изменениями.

Я часто встречаю заявления: «Если вы располагаете весь ваш код в контроллерах, вы нарушаете SRP!». Представим простое приложение, представляющее собой списки сущностей с возможностью создавать их, изменять и удалять. Так называемое CRUD-приложение. Все, что оно делает — лишь предоставляет интер-

фейс к строчкам в базе данных, без какой-либо дополнительной логики.

```
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');
}
```

Какие изменения возможны в таком приложении? Добавление/удаление полей или сущностей, косметические изменения интерфейса... сложно представить что-то большее. Я не думаю, что такой код нарушает SRP. Почти. Теоретически, может поменяться алгоритм редиректов, но это мелочи, я не вижу смысла рефакторить данный код.

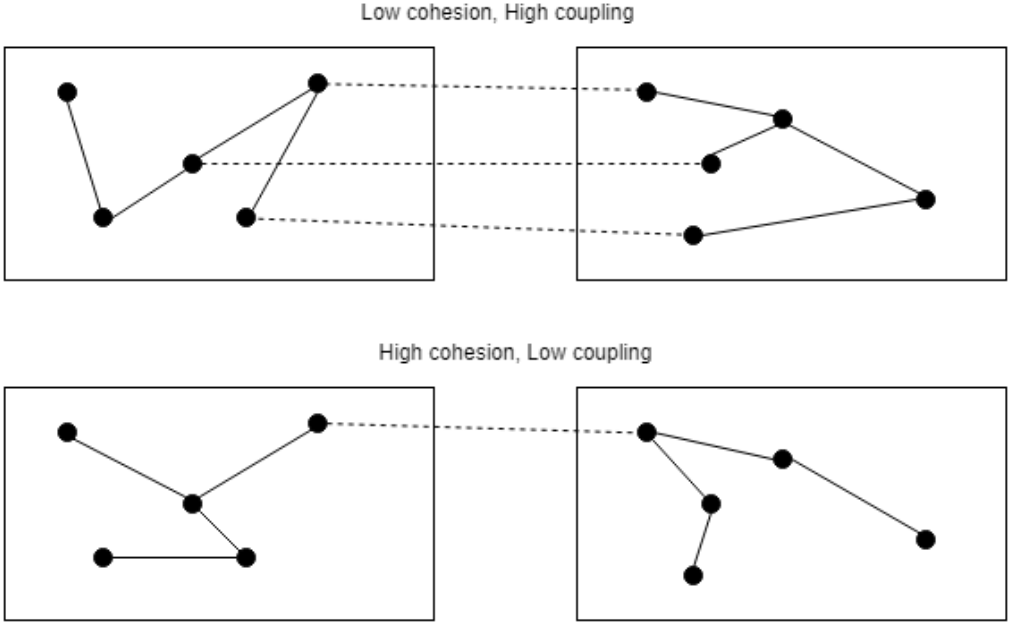
Рассмотрим новое требование к этому приложению из прошлой главы — загрузка аватара пользователя и посылка email после регистрации:

```
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');
}
```

Метод **store** содержит уже несколько ответственностей. Напомню, что кроме него есть ещё и метод **update**, меняющий сущность, и код, загружающий аватары, должен быть скопирован туда. Любое изменение в алгоритме загрузки аватар уже затронет как минимум два места в коде приложения. Часто бывает сложно уловить момент, когда стоит начать рефакторинг. Если эти изменения коснулись лишь сущности Пользователь, то, вероятно, этот момент ещё не настал. Однако, наверняка загрузка картинок понадобится и в других частях приложения.

Здесь я бы хотел остановиться и поговорить о двух важных базовых характеристиках программного кода — связность (cohesion)

и связанность (coupling). Связность — это степень того, как методы одного класса (или части другой программной единицы: функции, модуля) сконцентрированы на главной цели этого класса. Звучит очень похоже на SRP. Связанность между двумя классами (функциями, модулями) — это степень того, как много они знают друг о друге. Сильная связанность означает, что какие-то знания принадлежат нескольким частям кода и каждое изменение может вызвать каскад изменений в других частях приложения. Под знаниями я имею в виду какую-нибудь логику, например знание о том, как сохранить картинку в хранилище, или как обновить поля сущности Пользователь. То же самое, что и ответственность, но под другим углом.



Текущая ситуация с методом **store** является хорошей примером потери качества кода. Он начинает реализовывать несколько ответственностей — связность падает. Загрузка картинок реализована в нескольких местах приложения — связанность растет. Самое время вынести загрузку изображений в свой класс.

Первая попытка:

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

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

Я привёл этот пример, потому что я часто встречаю такое вынесение функционала, захватывающее слишком многое с собой. В этом случае класс **ImageUploader** кроме своей главной обязанности (загрузки изображений) присваивает и значение полю класса **User**. Что в этом плохого? Класс **ImageUploader** знает про класс **User** и его свойство **avatarUrl**. Такие знания часто меняются. Простейший случай — загрузка изображений для другой сущности. Чтобы реализовать это изменение, придётся изменять класс **ImageUploader**, а также методы класса **UserController**. Это и есть пример, когда одно маленькое изменение порождает целый каскад изменений в классах, не связанных с изначальным изменением.

Попробуем реализовать **ImageUploader** с высокой связностью:

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

Да, это не выглядит как логика, которую необходимо выносить, но в будущем загрузка изображений может стать сложнее (например, добавится создание миниатюр). Даже если этот код так и останется одной строчкой — функционал загрузки изображений вынесен в отдельный класс и много времени это не отняло. Любые изменения в будущем будет проще реализовать.

Dependency Injection

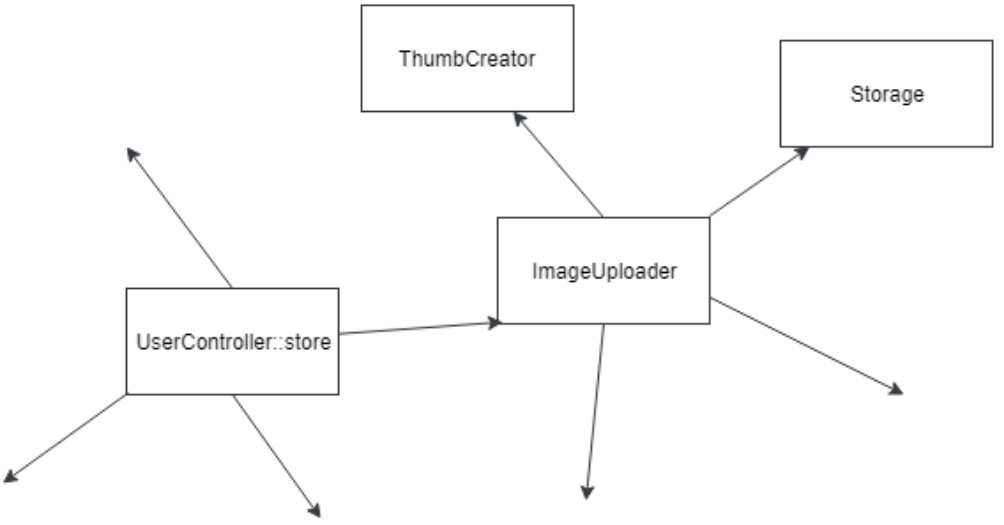
Класс **ImageUploader** создан, но как использовать его в методе **UserController::store**?

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

Или просто сделать метод **upload** статическим и вызывать его так:

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

Это выглядит красиво и легко, но теперь метод **store** имеет жесткую зависимость от класса **ImageUploader**. Представим, что таких зависимых методов в приложении стало много и команда решила использовать другое хранилище для изображений. Но не для всех, а лишь некоторых мест их загрузки. Как разработчики будут реализовывать это изменение? Создадут класс **AnotherImageUploader** и заменят вызовы класса **ImageUploader** на вызовы **AnotherImageUploader** во всех нужных местах. В крупных проектах такие изменения, затрагивающие большое количество классов, крайне нежелательны — они часто приводят к ошибкам. Изменение способа хранения изображений не должно влиять на код, работающий с сущностями.



Приложение с такими жёсткими связями выглядит как металлическая сетка. Очень сложно взять, например, класс **ImageUploader**, и использовать его в другом приложении. Или написать для него юнит-тесты.

Вместо жёстких зависимостей на классы техника **Внедрения Зависимостей** (Dependency Injection, DI) предлагает классам просить нужные зависимости.

```
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 и другие фреймворки содержат контейнер зависимостей (DI-контейнер) — специальный сервис, который берет на себя создание экземпляров нужных классов и внедрение их в зависящие классы. Метод **store** может быть переписан так:

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

В Laravel контейнер зависимостей умеет внедрять зависимости прямо в аргументы методов контроллеров. Зависимости стали менее жёсткими. Классы не создают другие классы и не требуют статических методов. Однако, метод **store** и класс **ImageUploader** зависят от конкретных классов. Принцип Инверсии Зависимостей (буква D в SOLID) гласит: > Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Требование абстракции в сообществах языка PHP и остальных, где есть понятие **интерфейс**, трактуется однозначно: зависимости должны быть только от интерфейсов, а не от классов. Я буду часто повторять, что проекты бывают разные. Для какого-нибудь проекта будет нормальным сразу заводить интерфейс, и потом класс, его реализующий. Для других же это будет не совсем оправданным.

Вы, вероятно, слышали про технику Test-driven Development (TDD). Грубо говоря, она предлагает писать тесты одновременно с кодом, который этими тестами проверяется. Попробуем рассмотреть написание класса **ImageUploader** одновременно с тестами. Классы **Storage** и **ThumbCreator**, необходимые для работы **ImageUploader** ещё не реализованы, но это не мешает реализовать его и проверить на соответствия требованиям. Можно создать интерфейсы **Storage** и **ThumbCreator** с необходимыми методами и начать работу. Для тестирования эти интерфейсы будут «реализованы» с помощью специальных объектов-моков (мы поговорим про них в главе про тестирование).

```
interface Storage
{
    // методы...
}

interface ThumbCreator
{
    // методы...
}

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(...
    }
}
```

Класс **ImageUploader** все ещё не может быть использован в приложении, поскольку для хранения сущностей и создания миниатюр есть только интерфейсы, без реализаций. Но он уже написан и протестирован. Позже, когда будут написаны классы, реализующие интерфейсы Storage и ThumbCreator, можно будет настроить контейнер зависимостей в Laravel, например, так:

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

После этого класс **ImageUploader** может быть использован в приложении. Когда контейнер зависимостей создаёт экземпляр класса, он сканирует его зависимости в конструкторе (**Storage** и **ThumbCreator**), находит нужные реализации (**S3Storage** и **ImagickThumbCreator**) исходя из своей конфигурации и подставляет их создаваемому объекту. Если эти реализации тоже имеют зависимости, то они также будут внедрены.

Техника TDD хорошо зарекомендовала себя во многих проектах, где она стала стандартом. Мне тоже нравится этот подход. Разрабатывая функционал одновременно тестируя его, получаешь мало с чем сравнимое удовольствие, ощущая насколько продуктивно получается работать. Однако, я крайне редко вижу его использование на проектах. Оно требует некоего уровня архитектурного мышления, поскольку необходимо заранее знать какие интерфейсы будут нужны, заранее декомпозировать приложение.

Обычно на проектах все намного проще и прозаичнее. Сначала пишется класс **ImageUploader**, который содержит всю логику по хранению и созданию миниатюр. Потом, возможно, некоторый функционал будет вынесен в классы **Storage** и **ThumbCreator**. Интерфейсы не используются. Иногда на проекте случается примечательное событие — один из разработчиков читает про Принцип Инверсии Зависимостей и решает, что у проекта серьезные проблемы с архитектурой. Классы не зависят от абстракций! Нужно немедленно создать интерфейс к каждому классу! Но имена **ImageUploader**, **Storage** и **ThumbCreator** уже заняты классами. Как правило, в данной ситуации выбирается один из двух кошмарных путей создания интерфейсов.

Первый это создание пространства имён `*\Contracts` и создание всех интерфейсов там. Пример из исходников Laravel:

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

Тут совершён двойной грех: использование одного имени для

интерфейса и класса, а также одного имени для абсолютно разных программных объектов - имя Repository для конфигурации и кеша. Механизм пространств имён даёт возможность для таких обходных маневров. Как можно увидеть, даже в коде класса приходится использовать алиасы **CacheContract** и **ConfigContract**. Любой Laravel проект имеет 4 программных объекта с именем **Repository**. В обычном приложении разработчики используют фасады **Cache** и **Config** или функции-хелперы, поэтому не сталкиваются с неудобствами, но если использовать DI, то класс, который использует и кеширование и конфигурацию, выглядит так (если не использовать алиасы):

```
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;
    }
}
```

Только имена переменных помогают понять какая конкретно зависимость была использована. Однако, имена Laravel-фасадов для этих интерфейсов выглядят весьма натурально: **Config** и **Cache**. С такими именами для интерфейсов, классы их использующие выглядели бы намного лучше.

Второй вариант: использование суффикса **Interface**, например, создать интерфейс **StorageInterface**. Таким образом, имея класс **Storage** реализующий интерфейс **StorageInterface**, мы постулируем, что у нас есть интерфейс и его «главная» реализация. Все остальные реализации этого интерфейса выглядят вторичными. Само существование интерфейса **StorageInterface** выглядит искусственным: он был создан, чтобы код удовлетворял каким-то принципам, или просто для юнит-тестирования.

Это явление встречается и в других языках. В C# принят префикс I*. Интерфейс **IList** и класс **List**, например. В Java не приняты префиксы и суффиксы, но там часто случается такое:

```
class StorageImpl implements Storage
```

Это тоже ситуация с интерфейсом и его реализацией по умолчанию.

Возможны две ситуации с интерфейсами:

1. Есть интерфейс и несколько возможных реализаций. В этом случае интерфейс стоит назвать естественным именем, а в именах реализаций использовать префиксы, которые описывают эту реализацию.

```
interface Storage{}

class S3Storage implements Storage{}

class FileStorage implements Storage{}
```

1. Есть интерфейс и одна реализация. Другую реализацию сложно представить. В этом случае интерфейс не нужен, можно просто использовать класс с естественным именем.

Если нет прямой необходимости использовать интерфейсы в проекте, то использование классов в механизме внедрения зависимостей вполне нормальная практика. Давайте снова взглянем на класс **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->...
    }
}
```

Он зависит от неких **Storage** и **ThumbCreator** и использует только их публичные методы. Разработчику, который работает в данный момент с этим кодом, всё равно классы это или интерфейсы. Контейнер внедрения зависимостей, убрав необходимость создавать объекты зависимостей, подарил нам супер-абстракцию: классам совсем неважно знать от чего именно он зависит - от интерфейса или класса. В любой момент, при изменении условий, класс может быть сконвертирован в интерфейс, а весь его функционал перенесен в новый класс, реализующий этот интерфейс

(**S3Storage**). Это изменение, наряду с конфигурацией контейнера зависимостей будут единственными на проекте. Весь остальной код, использовавший класс, будет работать как прежде, только теперь он зависит от интерфейса. Используя суффиксы **Interface** или другие отличительные признаки интерфейсов, мы лишаем себя этой абстракции, а также загрязняем наш код.

```
class Foo
{
    public function __construct(
        LoggerInterface $logger, StorageInterface $storage) {...}
}

class Foo
{
    public function __construct(
        Logger $logger, Storage $storage) {...}
}
```

Просто сравните эти два конструктора.

Разумеется, код публичной библиотеки или пакета должен быть максимально гибким. Поэтому, все зависимости в них должны быть на интерфейсы (или абстрактные классы). Однако, в обычном проекте зависимости на реальные классы — абсолютно нормальная практика.

Наследование

Наследование называют одним из трех «китов» ООП и разработчики обожают его. Однако, очень часто оно используется в неверном ключе, когда новый класс хочет использовать некий функционал, его наследуют от класса, который этот функционал имеет. Также часта ситуация, когда некоторой группе классов нужен один и тот же функционал, вместо использования DI, создается

некий Base* класс, в котором этот функционал реализуется, и все классы наследуются от него. Я приведу наполовину синтетический пример, который покажет насколько опасно использовать наследование подобным образом. В Laravel есть интерфейс **Queue**, содержащий методы работы с очередями и много классов, реализующих его. Допустим, наш проект использует **RedisQueue**.

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

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

Разумеется, классы проекта используют зависимость на интерфейс **Queue**, а менеджер зависимостей лишь подставляет **RedisQueue**. Когда возникла необходимость логировать все задачи в очереди, был создан класс **OurRedisQueue**, отнаследованный от **RedisQueue**.

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

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

Задача решена: все вызовы метода **push** логируются. Некоторое времени спустя, в новой версии Laravel, в интерфейсе **Queue** добавился метод **pushOn**, который представляет собой тот же **push**, но с другой очередностью параметров. Класс **RedisQueue** получил очевидную реализацию:

```
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)
    {
        // реализация
    }

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

Поскольку класс **OurRedisQueue** отнаследован от **RedisQueue**, никаких изменений в проекте при обновлении версии фреймворка не понадобилось. Всё работает как прежде и команда начала использовать новый метод **pushOn**.

В новом обновлении команда Laravel могла сделать небольшой рефакторинг:

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

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

    private function innerPush(...)
    {
        // реализация
    }
}
```

Рефакторинг абсолютно естественный и не меняет поведения класса, однако через некоторое время после обновления команда замечает, что логирование помещения сообщений в очередь работает не всегда. Легко предположить, что логирование метода **push** работает, а **pushOn** перестало.

Когда наследуются от неабстрактного класса, то у этого класса на высоком уровне образуются две ответственности: перед собственными клиентами и перед наследниками, которые тоже используют его функционал. Вторая ответственность не очень явная и довольно легко сделать ошибку, которая приведёт к сложным и трудноуловимым багам. Даже на таком простом примере с логированием это привело к багу, который было бы не так просто найти.

Чтобы избежать подобных осложнений, я в своих проектах все неабстрактные классы помечаю как final, запрещая наследование от них. Шаблон для создания нового класса в моей среде разработки содержит ключевые слова ‘final class’ вместо просто

‘class’. Финальные классы имеют ответственность только перед своими клиентами и её гораздо проще контролировать.

Кстати, дизайнеры языка программирования Kotlin, судя по всему, думают также и решили сделать классы в своём языке финальными по умолчанию. Чтобы сделать наследование возможным, необходимо использовать ключевое слово ‘open’ или ‘abstract’:

```
open class Foo {}
```

Мне нравится это.

Эта концепция — «final or abstract» — не полностью устраняет опасность наследования реализации. Абстрактный класс с protected-методами и его наследники могут попасть в такую же ситуацию, как описанную ранее с очередями. Каждое использование ключевого слова protected создаёт неявную связь между классами предка и наследника. Изменения в базовом классе могут породить баги в наследниках.

Механизм внедрения зависимостей предоставляет возможность просто просить нужный функционал, без необходимости наследоваться. Задача логирования сообщений в очереди может быть решена с помощью шаблона Декоратор:

```
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);
    }
}

// конфигурация контейнера зависимостей
// в сервис провайдере
$this->app->bind(Queue::class, LoggingQueue::class);

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

> Предупреждение: этот код не будет работать в реальном Laravel-окружении, поскольку эти классы имеют более сложную процедуру инициации.

Контейнер с такой конфигурацией будет подставлять экземпляр **LoggingQueue** каждому, кто просит экземпляр **Queue**. Экземпляры же класса **LoggingQueue** будут получать экземпляр **RedisQueue** и будут перенаправлять вызовы к нему, логируя их. После обновления Laravel с новым методом **pushOn** появится ошибка, что класс **LoggingQueue** не реализует все методы интерфейса **Queue**. Команда может решить как именно логировать этот метод и нужно ли.

Еще одним плюсом данного подхода является то, что конструктор классов полностью под контролем. В варианте с наследованием приходится вызывать `parent::__construct` и передавать туда все нужные параметры. Это станет дополнительной, совершенно

ненужной связью между двумя классами. Класс декоратора же не имеет никаких неявных связей с декорируемым классом и позволяет избежать целого ряда проблем в будущем.

Пример с загрузкой картинок

Вернемся к примеру с загрузкой картинок с предыдущей главы. Класс **ImageUploader** был вынесен из контроллера, чтобы реализовать логику загрузки изображений. Требования к этому классу:

- загружаемая картинка должна быть проверена на неприемлемый контент
- если проверка пройдена, картинка должна быть загружена в определенную папку
- если нет, то пользователь, который загрузил эту картинку, должен быть заблокирован после нескольких попыток.

```
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();

        // Проверки используя $this->googleVision,
        // $weakerRules и $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();
    }
}
```

Начальный рефакторинг

Простая ответственность за загрузку картинок разрослась и стала содержать несколько других ответственностей. Этот класс явно нуждается в рефакторинге.

Если представить, что класс **ImageUploader** будет вызываться из консоли, то **Auth::user()** будет возвращать null, поэтому должна быть добавлена соответствующая проверка, но гораздо удобнее и гибче просто просить в этом методе объект пользователя (**User** $uploadedBy) потому, что:

1. В этом случае можно быть уверенным, что в этой переменной будет не-null значение.
2. Каждый, кто вызывает этот класс, может сам решить какой объект пользователя ему передать. Это не всегда **Auth::user()**.

Функционал блокировки пользователя может понадобиться где-то еще. Сейчас это всего две строки кода, но в будущем там могут появиться отправка email и другие действия. Выделим отдельный класс для этого:

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

> Это действие часто встречает мощное противодействие со стороны других разработчиков. «Зачем делать целый класс ради двух строк?». «Теперь будет трудно читать код, ведь надо будет каждый раз искать этот новый класс в редакторе, чтобы посмотреть как все сделано». В книге потихоньку будут даваться частичные причины такого выноса логики в классы. Здесь же я могу лишь написать, что в современных IDE классы создаются за секунды, навигация осуществляется одним кликом, а название класса **BanUserCommand** быстро позволяет понять, что он делает, без заглядывания внутрь.

Следующая ответственность: «блокировка пользователя после нескольких попыток загрузить неподобающий контент». Параметр **$banThreshold** был добавлен в попытке добавить гибкости классу. Как часто случается, эта гибкость никому не оказалась нужной. Стандартное значение 5 всех устраивало. Проще это вынести в константу. Если в будущем эта гибкость понадобится, можно будет это добавить через конфигурацию или параметры фабрики.

```
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;
        }
    }
}
```

Реакция системы на загрузку неподобающего контента может поменяться в будущем, но эти изменения коснутся только этого класса. Эта локальность изменений, когда для изменения одной

логики не надо копаться в тонне другой, крайне важна для больших проектов.

Следующая ответственность, которую надо убрать, это «проверка контента картинок»:

```
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
    {
        // Проверки используя $this->googleVision,
        // $weakerRules и $imageContent
    }
}
```

```
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;
    }
}
```

Класс **ImageUploader** потерял несколько ответственностей и весьма рад этому. Он не заботится о том, как именно проверять картинки и что произойдёт, если там будет нарисовано что-то нехорошее. Он просто выполняет некую оркестрацию. Но мне все еще не нравятся параметры метода **upload**. Ответственности были вынесены в соответствующие классы, но их параметры все ещё здесь и вызовы этого метода до сих пор выглядят уродливо:

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

Булевы параметры всегда выглядят уродливо и повышают когнитивную нагрузку на чтение кода. Я попробую удалить их двумя разными путями:

- ООП путем;
- путём конфигурации.

ООП путь

Я собираюсь использовать **полиморфизм**, поэтому надо создать интерфейсы.

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

Создан интерфейс **ImageChecker** и три его реализации:

- **StrictImageChecker** для проверки картинок со строгими правилами.
- **WeakImageChecker** для нестрогой проверки.
- **TolerantImageChecker** для случаев, когда проверка не нужна.

WrongImageUploadsListener класс превратится в интерфейс с двумя реализациями:

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

final class BanningWrongImageUploadsListener
    implements WrongImageUploadsListener
{
    // реализация та же самая
    // с RateLimiter и BanUserCommand
}

final class EmptyWrongImageUploadsListener
    implements WrongImageUploadsListener
{
    public function handle(User $user)
    {
        // ничего не делаем
    }
}
```

Класс **EmptyWrongImageUploadsListener** будет использоваться вместо параметра **$dontBan**.

```
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;
    }
```

```
}
```

Логика булевых параметров переехала в интерфейсы и их реализации. Работа с файловой системой тоже может быть упрощена созданием фасада для работы с ней (я говорю о шаблоне **Facade**, а не о Laravel-фасадах). Единственная проблема, которая осталась, это создание экземпляров **ImageUploader** с нужными зависимостями для каждого случая. Она может быть решена комбинацией шаблонов **Builder** и **Factory**, либо конфигурацией DI-контейнера.

Признаться, я этот ООП путь привел лишь для того, чтобы показать, что «так тоже можно». Для текущего примера решение выглядит чересчур громоздким. Попробуем другой вариант.

Configuration way

Я буду использовать файл конфигурации Laravel, чтобы хранить все настройки. config/image.php:

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

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

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

Класс **ImageUploader**, использующий конфигурацию (интерфейс **Repository**):

```
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;
    }
}
```

Да, код не выглядит столь чистым как в ООП-варианте, но его конфигурация и реализация весьма просты. Для загрузки картинок этот вариант явно оптимальнее, но в других случаях с более сложной конфигурацией или оркестрацией, ООП-вариант будет

предпочтительнее.

Расширение интерфейсов

Иногда нам необходимо расширить интерфейс каким-нибудь методом. В главе про Доменный слой мне нужно будет отправлять несколько событий в каждом классе сервиса. Интерфейс **Dispatcher** в Laravel имеет только метод **dispatch**, обрабатывающий одно событие:

```
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);
}
```

Каждый раз придется делать такой **foreach**:

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

Но копипастить это в каждом методе сервисов не очень хочется. Языки C# и Kotlin имеют фичу «метод-расширение», который натурально «добавляет» метод к любому классу или интерфейсу:

```
fun Dispatcher.multiDispatch(events: Collection<Event>) {
    events.forEach { dispatch(it) }
}
```

После этого, каждый класс может использовать метод **multiDispatch**:

```
dispatcher.multiDispatch(events);
```

В PHP такой фичи нет. Для интерфейсов, принадлежащих вашему проекту, метод может быть добавлен в интерфейс и в классы, его реализующие. В случае абстрактного класса, метод может быть добавлен в него и классы-наследники менять не придется (кстати, поэтому я стал больше предпочитать абстрактные классы). Для интерфейсов, не принадлежащих проекту, это невозможно, поэтому обычное решение такое:

```
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);
    }
}
```

Использование наследования для того, чтобы унаследовать функционал — не очень хорошая идея. Конструкторы становятся более сложными со всеми этими **parent::__construct** вызовами. Расширение другого интерфейса этим же базовым классом повлечет за собой изменения конструкторов всех сервисов.

Создание нового интерфейса выглядит более естественным. Классы сервисов нуждаются только в одном методе **multiDispatch** и можно сделать простой интерфейс с этим методом:

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

и реализовать его:

```
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);
```

```
    }
}
```

Класс **BaseService** может быть удалён и сервис-классы могут просто использовать этот новый интерфейс:

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

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

    public function someMethod()
    {
        //...

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

В качестве бонуса теперь можно легко переключиться на другой движок обработки событий вместо стандартной Laravel-реализации, реализовав этот интерфейс в новом классе, используя там вызов другого движка. Такие интерфейсы-обёртки могут быть весьма полезны, делая проект менее зависимым от конкретных библиотек.

Когда вызывающие классы хотят полный интерфейс, просто с новым методом, новый интерфейс может наследоваться от старого:

```
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
}
```

Класс здесь реализует недостающий метод и делегирует все остальные другой реализации. Для больших интерфейсов это может быть весьма долгим, рутинным действием. Здесь я могу опять привести в пример язык Kotlin, в котором делегирование простым ключевым словом **by**. Я надеюсь, в PHP тоже появится что-то подобное.

Трейты

Трейты в PHP позволяют магически добавлять функционал в класс практически бесплатно. Это весьма мощная магия: они могут залезать в приватные части классов и добавлять новые публичные и даже приватные методы в них. Я не люблю их. Это часть тёмной магии PHP, мощной и опасной. Они могут с успехом использоваться в классах тестов, поскольку там нет хорошей причины организовывать полноценное DI, но лучше избегать их использования в главном коде приложения. Трейты — это не ООП, а чистые ООП решения всегда будут более естественными.

Трейты, расширяющие интерфейсы

Проблема с множественной обработкой событий может быть решена с помощью трейта:

```
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);
    }
}
```

Трейт **MultiDispatch** предполагает, что у класса, который будет его использовать есть поле **dispatcher** класса **Dispatcher**. Лучше не делать таких неявных предположений. Решение с отдельным интерфейсом **MultiDispatcher** намного более явное и стабильное.

Трейты как части класса

В языке C# имеется такая фича как **partial class**. Она может быть использована когда некоторый класс становится большим и разработчик может разделить его на несколько файлов:

```
// 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();
```

Когда то же самое случается в PHP, трейты используются с той же целью. Пример из Laravel:

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

Большой класс **Request** разделен на несколько составляющих. Когда класс «хочет» разделиться на несколько — это большой намек на то, что у него слишком много ответственностей. Класс **Request** вполне можно было бы скомпоновать из нескольких других классов, таких как **Session**, **RequestInput**, **Cookies** и т.д.

```
class Request
{
    /** @var Session */
    private $session;

    /** @var RequestInput */
    private $input;

    /** @var Cookies */
    private $cookies;

    //...

    public function __construct(
        Session $session,
        RequestInput $input,
        Cookies $cookies
        //...
    ) {
        $this->session = $session;
        $this->input = $input;
        $this->cookies = $cookies;
        //...
    }
}
```

Вместо того чтобы комбинировать класс из трейтов, намного лучше разбить ответственности класса и использовать этот класс как комбинацию этих ответственностей. Настоящий конструктор класса **Request** вполне недвусмысленно на это намекает:

```
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)
    {
        //...
    }

    //...
}
```

Трейты как поведение

Eloquent-трейты, такие, как **SoftDeletes** - примеры поведенческих трейтов. Они изменяют поведение классов. Классы Eloquent моделей содержат как минимум две ответственности: хранение состояния сущности и выборку/сохранение/удаление сущностей из базы, поэтому Eloquent-трейты тоже могут менять то, как модели взаимодействуют с базой данных, а также добавлять новые поля и методы в них. Эти трейты надо как-то конфигурировать и тут раскрывается большой простор для фантазии разработчиков пакетов. Трейт **SoftDeletes**:

```
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';
    }
}
```

Он ищет константу **DELETED_AT** в классе и если находит, то использует её значение для имени поля, либо использует стандартное. Даже для такой простейшей конфигурации была применена магия (функция **defined**). Другие Eloquent трейты имеют более сложную конфигурацию. Я нашел одну библиотеку и Eloquent трейт там выглядит так:

```
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;
    }
}
```

Простая настройка, а сколько сложностей. Просто представьте:

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

Явная настройка поведения с удобной конфигурацией, которую будет подсказывать ваша среда разработки, без замусоривания исходного класса. Идеально! Поля и отношения в Eloquent виртуальные, поэтому эта реализация поведений тоже возможна. Без магии, конечно, не обойдётся, это все-таки Eloquent, но выглядит намного более объектно-ориентированно и, что намного важнее, явно.

Разумеется, эти behaviours существуют только в моем воображении и, вероятно, я не вижу некоторых проблем, но эта идея мне нравится намного больше, чем трейты.

Бесполезные трейты

Некоторые трейты просто абсолютно бесполезны. Я нашел один такой в исходниках Laravel:

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

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

Он просто добавляет методы **Dispatcher** в класс.

```
class WantsToDispatchJobs
{
    use DispatchesJobs;

    public function someMethod()
    {
        //...

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

Но ведь намного проще делать так:

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

        \Bus::dispatch(...);

        //or just

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

И данный трейт просто не нужен. Замечу также, что эта простота — главная причина того, что разработчики не используют внедрение зависимостей в PHP.

```
class WantsToDispatchJobs
{
    /** @var Dispatcher */
    private $dispatcher;

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

    public function someMethod()
    {
        //...

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

Этот класс намного проще прошлых примеров, поскольку имеет явную зависимость от **Dispatcher**. Он явно постулирует, что для работы ему необходим диспатчер. В случае, когда этот класс захотят перенести в другой проект или написать тесты для него, разработчикам не придется полностью сканировать его код и искать эти глобальные вызовы функций или фасадов. Единственная проблема — громоздкий синтаксис с конструктором и приватное поле. Синтаксис в языке Kotlin намного более элегантный:

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

Синтаксис PHP является некоторым барьером для использования DI и я надеюсь, что скоро это будет нивелировано либо изменениями в синтаксисе, либо инструментами в средах разработки.

Я писал эти строки в начале 2019-го года и не предполагал, что разработчики языка PHP будут так быстры в этом. В современном PHP можно писать так:

```
class WantsToDispatchJobs
{
    public function __construct(private Dispatcher $dispatcher) {}

    public function someMethod()
    {
        //...

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

Теперь сложно придумать причину НЕ использовать DI в PHP-проектах.

После нескольких лет использования и неиспользования трейтов я могу сказать, что разработчики используют трейты по двум причинам:

- борясь с последствиями архитектурных проблем;
- создавая архитектурные проблемы (иногда не осознавая этого).

Надо лечить болезнь, а не симптомы, поэтому лучше найти причины, заставляющие нас использовать трейты и постараться их исправить.

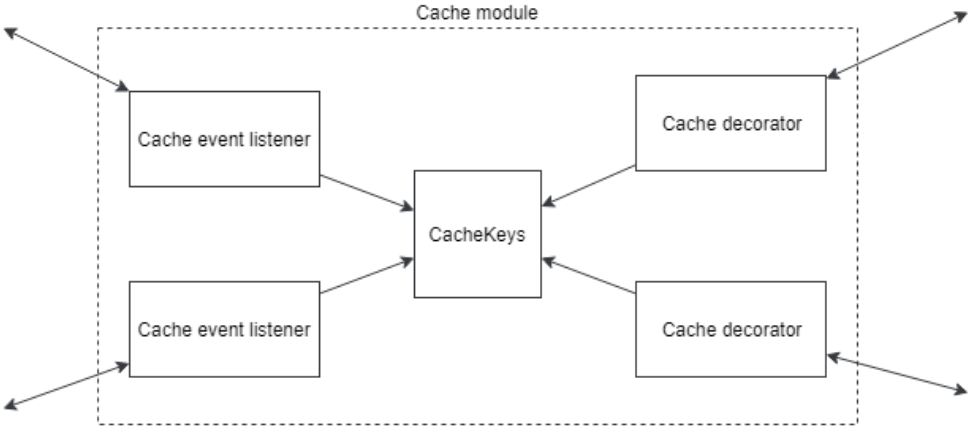
Статические методы

Я писал, что используя статические методы и классы мы создаем жёсткую связь, но иногда это нормально. Пример из прошлой главы:

```
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);
```

Ключи кэширования необходимы в двух местах: в классах декораторах для выборки сущностей и в классах-слушателей событий, которые отлавливают события изменения сущностей и чистят кэш.

Я мог бы использовать класс **CacheKeys** через DI, но в этом мало смысла. Все эти классы декораторов и слушателей формируют некую структуру, которую можно назвать «модуль кэширования» для этого приложения. Класс **CacheKeys** будет приватной частью этого модуля. Никакой другой код приложения не должен об этом классе знать.



Использование статических методов для таких внутренних зависимостей, которые не работают с внешним миром (файлами, БД или API) — нормальная практика.

Пара слов в конце главы

Самое большое преимущество использования внедрения зависимостей это явный и чёткий контракт класса. Публичные методы говорят о том, какую работу способен этот класс исполнять. Параметры конструктора говорят о том, что нужно для этой работы этому классу. В больших, долго длящихся проектах это очень важно. Классы могут быть легко протестированы и использованы

в любых условиях. Необходимо лишь предоставить им нужные зависимости. Вся эта магия, как методы **__call**, фасады Laravel и трейты разрушают эту гармонию.

С другой стороны, мне сложно представить, например, HTTP-контроллеры вне приложения Laravel и, я надеюсь, никто не пишет юнит-тесты для них. Поэтому, это вполне подходящее место для использования функций-хелперов (**redirect()**, **view()**) и Laravel фасадов (**Response**, **URL**).

4. Безболезненный рефакторинг

> “Надежно зафиксированный больной не нуждается в анестезии.”

“Статическая” типизация

Маленькие и большие программные проекты отличаются многим, в том числе и стилем работы. Большую часть времени в небольших проектах занимается собственно кодирование - набор кодовой базы. В больших проектах же - навигация по этой кодовой базе: перемещение от одного класса к другому, от вызова метода к его коду, от кода метода к его вызовам (функция Find Usages). Иногда больше 95% времени на задачу занимает именно блуждание по лабиринту кода.

А для того, чтобы это блуждание было менее болезненным, проекты постоянно нуждаются в рефакторинге: - извлечение методов и классов из других методов и классов; - переименование их, добавление и удаление параметров;

Современные среды разработки (IDE) имеют на борту кучу инструментов для продвинутой навигации и рефакторинга, которые облегчают его, а иногда и выполняют его полностью автоматически. Однако, динамическая природа PHP часто вставляет палки в колёса.

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

// или

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

В обоих этих случаях IDE не может самостоятельно понять, что был вызван метод **publish** класса **Post**. Для того чтобы добавить новый параметр в этот метод, нужно будет найти все использования этого метода.

```php
public function publish(User $publishedBy)
```

IDE не сможет сама найти их. Разработчику придётся искать по всему проекту слово «publish» и найти среди результатов именно вызовы данного метода. Для каких-то более распространённых слов (name или create) и при большом размере проекта это может быть весьма мучительно.

Представим ситуацию когда команда обнаруживает, что в поле `email` в базе данных находятся значения, не являющиеся корректными email-адресами. Как это произошло? Необходимо найти все возможные присвоения полю **email** класса **User** и проверить их. Весьма непростая задача, если учесть, что поле `email` виртуальное и оно может быть присвоено вот так:

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

Эта автомагия, которая помогала нам так быстро создавать приложения, показывает свою истинную сущность, преподнося такие вот сюрпризы. Такие баги в продакшене иногда очень критичны и каждая минута важна, а я до сих пор помню, как целый день в огромном проекте, который длится уже лет 10, искал все возможные присвоения одного поля, пытаясь найти, где оно получает некорректное значение.

После нескольких подобных случаев тяжелого дебага, а также сложных рефакторингов, я выработал себе правило: делать PHP-код как можно более статичным. IDE должна знать все про каждый метод и каждое поле, которое я использую.

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

// или с phpDoc

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

Комментарии phpDoc могут помочь и в сложных случаях:

```
/**
 * @var Post[] $posts
 */
$posts = Post::all();
foreach($posts as $post) {
    $post->// Здесь IDE должна подсказывать
          // все методы и поля класса Post
}
```

Подсказки IDE приятны при написании кода, но намного важнее, что подсказывая их, она понимает откуда они и всегда найдёт их использования.

Если функция возвращает объект какого-то класса, он должен быть объявлен как return-тип (начиная с PHP7) или в `@return` теге phpDoc-комментария функции.

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

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

Меня пару раз спрашивали: зачем я делаю Java из PHP? Это не совсем так. Я просто создаю маленькие комментарии, чтобы иметь удобные подсказки от IDE прямо сейчас и огромную помощь в будущем, при навигации, рефакторинге и дебаггинге. Даже для небольших проектов они невероятно полезны.

Шаблоны

На сегодняшний день все больше и больше проектов имеют только API-интерфейс, однако количество проектов, напрямую генерирующих HTML все ещё велико. Они используют шаблоны, в которых много вызовов методов и полей. Типичный вызов шаблона в Laravel:

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

Он выглядит как вызов некоей функции. Сравните с этим псевдокодом:

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

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

Хочется так же описать и параметры шаблонов. Это легко, когда шаблоны написаны на чистом PHP — комментарии phpDoc легко бы помогли. Для шаблонных движков, таких как Blade, это не так просто и зависит от IDE. Я работаю в PhpStorm, поэтому могу говорить только про него. С недавних пор там тоже можно объявлять типы через phpDoc:

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

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

Я понимаю, что многим это кажется уже перебором и бесполезной тратой времени, но после всех этих усилий по статической «типизации» мой код в разы более гибкий. Я легко нахожу все использования полей и методов, могу переименовать все автоматически. Каждый рефакторинг приносит минимум боли.

Поля моделей

Использование магических методов **__get**, **__set**, **__call** и других соблазнительно, но опасно — находить такие магические вызовы будет сложно. Если вы используете их, лучше снабдить эти классы нужными phpDoc комментариями. Пример с небольшой Eloquent моделью:

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

Этот класс имеет несколько виртуальных полей, представляющих поля таблицы **users**, а также поле **roles**. С помощью пакета

laravel-ide-helper можно автоматически сгенерировать phpDoc для этого класса. Всего один вызов artisan команды и для всех моделей будут сгенерированы комментарии:

```
/**
 * 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->// Здесь IDE подскажет все поля!
```

Возвратимся к примеру из прошлой главы:

```
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');
}
```

Создание сущности User выглядит странновато. До некоторых изменений оно выглядело хотя бы красиво:

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

Потом пришлось его поменять, поскольку поле **avatarUrl** нельзя присваивать напрямую из объекта запроса.

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

Оно не только выглядит странно, но и небезопасно. Этот метод используется в обычной регистрации пользователя. В будущем

может быть добавлено поле **admin**, которое будет выделять администраторов от обычных смертных. Какой-нибудь сообразительный хакер может просто сам добавить новое поле в форму регистрации:

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

Он станет администратором сразу же после регистрации. По этим причинам некоторые эксперты советуют перечислять все нужные поля (есть ещё метод **$request->validated()**, но его изъяны будут понятны позже в книге, если будете читать внимательно):

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

Но если мы и так перечисляем все поля, может просто сделаем создание объекта более цивилизованным?

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

Этот код уже можно показывать в приличном обществе. Он будет понятен любому PHP-разработчику. IDE теперь всегда найдёт, что в этом месте полю **email** класса **User** было присвоено значение.

«Что, если у сущности 50 полей?» Вероятно, стоит немного поменять интерфейс пользователя? 50 полей - многовато для любого, будь то пользователь или разработчик. Если не согласны, то дальше в книге будут показаны пару приемов, с помощью которых можно сократить данный код даже для большого количества полей.

Laravel Idea

Это было настолько важным для меня, что я разработал плагин для PhpStorm - Laravel Idea[1]. Он весьма неплохо разбирается в магии Laravel и устраняет необходимость во всех phpDoc, о которых я писал выше, предлагает кучу кодо-генераций и сотни других функций. Приведу пару примеров.

```
User::where('email', $email);
```

Плагин виртуально свяжет строку `'email'` в этом коде с полем `$email` класса User. Это позволяет подсказывать все поля сущности для первого аргумента метода `where`, а также находить все подобные использования поля `$email` и даже автоматически переименовать все такие строки, если пользователь переименует `$email` в какой-нибудь `$firstEmail`. Это работает даже для сложных случаев:

```
Post::with('author:email');

Post::with([
    'author' => function (Builder $query) {
        $query->where('email', 'some@email');
    }]);
```

В обоих этих случаях PhpStorm найдёт, что здесь было использовано поле `$email`. То же самое с роутингом:

```
Route::get('/', 'HomeController@index');
```

Здесь присутствуют ссылки на класс `HomeController` и метод `index` в нём. Если попросить PhpStorm найти места, где используется метод index - он найдет место в этом файле роутов. Такие фичи,

[1] https://laravel-idea.com/?utm_medium=book&utm_source=book_architecture&utm_campaign=link_inside_book

на первый взгляд не нужные, позволяют держать приложение под бОльшим контролем, который просто необходим для приложений среднего или большого размеров.

Мы сделали наш код более удобным для будущих рефакторингов или дебага. Эта «статическая типизация» не является обязательной, но она крайне полезна. Необходимо хотя бы попробовать.

5. Слой Приложения

> “Я Винстон Вульф. Я решаю проблемы.”

Продолжаем наш пример. Приложение растёт, и в форму регистрации добавились новые поля: дата рождения и опция согласия получения email-рассылки.

```
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');
}
```

Потом у приложения появляется API для мобильного приложения и регистрация пользователей должна быть реализована и там. Давайте ещё нафантазируем некую консольную artisan-команду, которая импортирует пользователей и она тоже хочет их регистрировать. И telegram-бот! В итоге у приложения появилось несколько интерфейсов, кроме стандартного HTML и везде необходимо использовать такие действия, как регистрация пользователя или публикация статьи. Самое естественное решение здесь выделить общую логику работы с сущностью (User в данном примере) в отдельный класс. Такие классы часто называют сервисными классами:

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

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

Но множество интерфейсов приложения (API, Web, и т.д.) не являются единственной причиной создания сервисных классов. Методы контроллеров начинают расти и обычно содержат две большие части:

```php
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()->...
}
```

Этот метод реализует как минимум две ответственности: логику работы с HTTP запросом/ответом и бизнес-логику. Каждый раз когда разработчик меняет http-логику, он вынужден читать много кода бизнес-логики и наоборот. Такой код сложнее дебажить и рефакторить, поэтому вынесение логики в сервис-классы тоже может быть хорошей идеей для этого проекта.

Передача данных запроса

Начнем создавать класс **UserService**. Первой проблемой будет передача данных запроса туда. Некоторым методам не нужно много данных и, например, для удаления статьи нужен только её id. Однако, для таких действий, как регистрация пользователя, данных может понадобиться много. Мы не можем использовать класс **Request**, поскольку он доступен только для web и недоступен, например для консоли. Попробуем простые массивы:

```
final class UserService
{
    public function __construct(
        private ImageUploader $imageUploader,
        private 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;
    }
}

// Controller
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');
}
```

Я просто вынес логику без каких-либо изменений и вижу проблему. Когда мы попытаемся зарегистрировать пользователя из консоли, то код будет выглядеть примерно так:

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

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

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

Выглядит чересчур витиевато. Вместе с кодом создания пользователя в сервис переехала и HTML-специфика передаваемых данных. Булевы значения проверяются присутствием нужного ключа в массиве. Значения даты получаются преобразованием из строк. Использование такого формата данных весьма неудобно, если эти данные пришли не из HTML-формы. Нам нужен новый формат данных, более естественный, более удобный. Шаблон **Data Transfer Object**(DTO) предлагает просто создавать объекты с нужными полями:

```
final class UserCreateDto
{
    private string $email;

    private DateTime $birthDate;

    private bool $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;
    }
}
```

Частенько я слышу возражения вроде «Я не хочу создавать целый класс только для того, чтобы передать данные. Массивы справляются не хуже.» Это верно отчасти и для какого-то уровня сложности приложений создавать классы DTO, вероятно, не стоит. Дальше в книге я приведу пару аргументов в пользу DTO. Здесь же лишь напишу, что в современных IDE такой класс создается очень быстро - достаточно лишь описать поля - параметры конструктора и getter методы генерируются автоматически. Заглядывать в этот класс разработчики будут крайне редко, поэтому какой-то сложности в поддержке приложения он не добавляет. С появлением в современном PHP readonly полей и классов создавать такие DTO-классы стало еще легче.

```
readonly final class UserCreateDto
{
    public function __construct(
        public string $email;
        public DateTime $birthDate;
        public bool $subscribed;
    ) {}
}
```

Комбинация из private поля и геттер-метода использовалась, чтобы обеспечить неизменяемость данных внутри DTO-объекта. Модификатор readonly гарантирует её, поэтому мы можем сделать поля public.

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

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

        $user = new User();
        $user->email = $request->email;
        $user->avatarUrl = $avatarFileName;
        $user->subscribed = $request->subscribed;
        $user->birthDate = $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',
    ]);

    $dto = new UserCreateDto(
```

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

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

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

Теперь это выглядит канонично. Сервисный класс получает чистую DTO и выполняет действие. Правда, метод контроллера теперь довольно большой. Конструктор DTO класса может быть весьма длинным. Можно попробовать вынести логику работы с данными запроса оттуда. В Laravel есть удобные классы для этого — **Form Requests**.

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

    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');
    }
}
```

Если какой-либо класс просит класс, наследованный от **FormRequest**, как зависимость, Laravel создаёт его и выполняет валидацию автоматически. В случае неверных данных метод **store** не будет выполняться, поэтому можно всегда быть уверенными в валидности данных в **UserCreateRequest**.

Работа с базой данных

Простой пример:

```php
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();
    }
}
```

Публикация статьи — это пример простейшего не-CRUD действия. В примере все выглядит неплохо, но если попробовать вызвать метод сервиса из консоли, то нужно будет опять доставать сущность **Post** из базы данных.

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

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

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

Это пример нарушения Принципа единственной ответственности (SRP) и высокой связанности (coupling). Каждая часть приложения (и сервисные классы, и веб-контроллеры, и консольные команды) работает с базой данных. Любое изменение, связанное с работой с базой данных, может повлечь изменения во всем приложении. Необходимо спрятать работу с базой данных внутри сервисного класса. Иногда я вижу интересный вариант в виде метода **getById** в сервисах:

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

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

        if (!$postService->publish($post)) {
```

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

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

Этот код просто получает сущность и передаёт её в другой метод сервиса, но методу контроллера сущность **Post** не нужна. Он может просто попросить сервис опубликовать статью с таким то **id**. Наиболее логичное и простое решение:

```
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();
    }
}
```

Одно из главных преимуществ создания сервисных классов — это консолидация всей работы с бизнес-логикой и инфраструктурой, включая хранилища данных, такие как базы данных и файлы, в одном месте, оставляя Web, API, Console и другим интерфейсам работу исключительно со своими обязанностями. Часть приложения для работы с Web(веб-контроллеры, request классы) должна просто готовить данные для сервис-классов и показывать результаты пользователю. То же самое про другие интерфейсы. Это Принцип единственной ответственности для слоёв. Слоёв? Да.

Слоем называют группу классов, объединенных схожей ответственностью и схожими зависимостями. Например, слой веб-контроллеров - классы, которые принимают веб-запрос, передают его в сервисные классы, получают от них ответ и отдают результат в нужной форме. Слоем они названы из-за того, что веб-запрос(или запрос из консоли) проходит сквозь эти слои(веб-контроллеров, сервисный классов, работы с базой данных) и возвращается через них же.

Все сервис-классы, прячущие логику приложения внутри себя, формируют структуру, которая имеет множество имён: * **Сервисный слой** (Service layer), из-за **сервисных** классов. * **Слой приложения** (Application layer), потому что он содержит всю логику приложения, исключая интерфейсы к нему. * в GRASP этот слой называется **Слой контроллеров** (Controllers layer), поскольку сервисные классы там называются контроллерами. * наверняка есть и другие названия.

В этой книге я буду называть его **Слоем приложения**. Потому что могу.

То, что я описал здесь, очень похоже на архитектурный шаблон **Гексагональная архитектура**, или **Луковая архитектура**, или

еще десятки подобных. Неудивительно, поскольку они решают ровно те же задачи. Однако, гораздо полезнее для развития разработчика самому осознать причины выделения кода в отдельные классы, почувствовать какие части кода могут работать с базой данных или данными веб-запроса. Самому увидеть как участки кода с похожими обязанностями и потребностями выстраиваются в некое подобие слоев. Самому чувствовать когда проекту требуется выделение логики в другие классы, а когда нет. Получив подобный опыт и набив руку, можно ознакомиться с данными шаблонами и, возможно, принять один из них как стандарт на каком-либо проекте, полностью осознавая, что данный шаблон подходит проекту, не являясь микроскопом, коим забивают гвозди или пушкой, которой стреляют по воробьям.

Сервисные классы или классы команд?

Когда сущность большая, с кучей различных действий, сервисный класс для нее тоже будет большим. Разные действия, такие как отредактировать статью или опубликовать её, требуют разных зависимостей. Всем нужна база данных, но одни хотят посылать письма, другие — залить файл в хранилище, третьи — вызвать какое-нибудь внешнее API. Количество параметров конструктора сервисного класса растёт довольно быстро, хотя каждое из них может использоваться лишь в одном-двух методах. Довольно быстро становится понятно, что класс занимается слишком многими делами. Поэтому разработчики часто начинают создавать классы на каждое действие с сущностями.

Насколько мне известно, стандарта на название таких классов тоже нет. Я видел суффикс **UseCase** (например **PublishPostUseCase**), суффикс **Action** (**PublishPostAction**), но предпочитаю суффикс **Command**: **CreatePostCommand**, **PublishPostCommand**, **DeletePostCommand**.

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

В шаблоне **Command Bus** суффикс **Command** используется для DTO-классов, а классы исполняющие команды называются **CommandHandler**.

```
readonly final class ChangeUserPasswordCommand
{
    public function __construct(
        public int $id;
        public string $oldPassword;
        public string $newPassword;
    ) {}
}

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

// или если один класс исполняет много команд

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

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

В книге я, для простоты, буду использовать **Service**-классы.

Небольшая ремарка про длинные имена классов. «**ChangeUserPasswordCo
— ого! Не слишком ли длинное имя? Не хочу писать его каждый раз!» Полностью его написать придется всего один раз — при создании. Дальше IDE будет полностью подсказывать его. Хорошее, полностью описывающее класс, название намного важнее. Каждый разработчик может сразу сказать что примерно делает класс **ChangeUserPasswordCommandHandler** не заглядывая внутрь него.

Пара слов в конце главы

Выделение слоя приложения - весьма ответственный шаг и причина должна быть серьезной. Их всего две:

1. Разные интерфейсы к одним и тем же действиям (Web, API, Console, различные боты). Тут вынесение общей логики просится само.
2. Разросшиеся и сложные логики обработки веб-запроса, например, и бизнес-логики. В данном случае разделение логик по разным местам может заметно улучшить связность кода. Как правило, это ведет к повышенной багостойкости кода.

6. Обработка ошибок

> “Каждый заслуживает второй второй шанс, Пэм!”

Язык C, который дал основу синтаксиса для многих современных языков, имеет простую конвенцию для ошибок. Если функция должна вернуть какие-то данные, но не может вернуть из-за ошибки, она возвращает null. Если функция выполняет какую-то задачу, не возвращая никакого результата, то в случае успеха она возвращает 0, а в случае ошибки -1 или какой-нибудь код ошибки. Много PHP разработчиков полюбили такую простоту и используют те же принципы. Код может выглядеть так:

```
readonly final class ChangeUserPasswordDto
{
    public function __construct(
        public readonly int $userId,
        public readonly string $oldPassword,
        public readonly string $newPassword)
    {}
}

final class UserService
{
    public function changePassword(
        ChangeUserPasswordDto $dto): bool
    {
        $user = User::find($dto->userId);
        if($user === null) {
            return false; // пользователь не найден
```

```
        }

        if(!password_verify($dto->oldPassword, $user->password)) {
            return false; // старый пароль неверный
        }

        $user->password = password_hash($dto->newPassword);
        return $user->save();
    }
}

final class UserController
{
    public function changePassword(UserService $service,
                        ChangeUserPasswordRequest $request)
    {
        if($service->changePassword($request->getDto())) {
            // возвращаем успешный ответ
        } else {
            // возвращаем ответ с ошибкой
        }
    }
}
```

Ну, по крайней мере, это работает. Но что, если пользователь хочет узнать в чем причина ошибки? Комментарии рядом с `return false` бесполезны во время выполнения кода. Можно попробовать коды ошибки, но часто кроме кода нужна и дополнительная информация для пользователя. Попробуем создать специальный класс результата функции:

```
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;
    }
}
```

Конструктор этого класса приватный, поэтому все объекты могут быть созданы только с помощью статических

методов **FunctionResult::success** и **FunctionResult::error**. Этот простенький трюк называется “именованные конструкторы”.

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

Выглядит намного проще и информативнее, чем

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

Как только конструктор вашего класса вырастает так, что вызовы new этого класса выглядят коряво, задумайтесь об именованных конструкторах для него. Наш код будет выглядеть так:

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

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

        $user->password = password_hash($dto->newPassword);

        $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) {
            // возвращаем успешный ответ
        } else {
            // возвращаем ответ с ошибкой
            // с текстом $result->errorMessage
        }
    }
}
```

Каждый метод (даже save() у Eloquent модели в этом воображаемом мире) возвращает объект **FunctionResult** с полной информацией о том, как завершилось выполнение функции. Когда я показывал этот пример на одном семинаре один слушатель сказал: “Зачем так делать? Есть же исключения!” Да, исключения (exceptions) есть, но дайте лишь показать пример из языка Go:

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

Обработка ошибок там реализована похожим образом. Без исключений. Популярность языка растёт, поэтому без исключений вполне можно жить. Однако, чтобы продолжать использовать

класс **FunctionResult**, придётся реализовать стек вызовов функций, необходимый для отлавливания ошибок в будущем и корректное логирование каждой ошибки. Все приложение будет состоять из проверок **if($result->success)**. Не очень похоже на код моей мечты… Мне нравится код, который просто описывает действия, не проверяя состояние ошибки на каждом шагу. Попробуем использовать исключения.

Исключения (Exceptions)

Когда пользователь просит приложение выполнить действие (зарегистрироваться или отменить заказ), приложение может выполнить его или нет. Во втором случае, причин может быть множество. Одним из лучших иллюстраций этого является список кодов HTTP-ответа. Там есть коды 2xx и 3xx для успешных ответов, таких как 200 Ok или 302 Found. Коды 4xx и 5xx нужны для неуспешных ответов, но они разные.

- 4xx для ошибок клиента: 400 Bad Request, 401 Unauthorized, 403 Forbidden, и т.д.
- 5xx для ошибок сервера: 500 Internal Server Error, 503 Service Unavailable, и т.д.

Соответственно, все ошибки валидации, авторизации, не найденные сущности и попытки изменить пароль с неверным старым паролем - это ошибки клиента. Недоступность стороннего API, ошибка хранилища файлов или проблемы со связью с базой данных - это ошибки сервера.

Есть две противоборствующие школы обработок ошибок:

1. Девизом школы Аскетов Исключения является “Исключения только для исключительных ситуаций”. Любое исключение считают вещью весьма необычной, способной произойти

только из-за событий непреодолимой силы (отказ бд или файловой системы) и почти все исключения превращаются в 500-тые ответы сервера. Для ситуаций с неверно введённым email или неправильным паролем они используют что-то вроде объекта **FunctionResult**.
2. Адепты же школы Единого Верного Пути считают любую негативную ситуацию, т.е. ситуацию, которая не даёт выполнить действие пользователя, исключением.

Код аскетов, как и их девиз, выглядит более логично, но ошибки клиента придётся постоянно протаскивать наверх, как в примерах выше, из функций к тем, кто их вызвал, из Слоя приложения в контроллеры и т.д. Код же их противников имеет унифицированный алгоритм работы с любой ошибкой (просто выбросить исключение) и более чистый код, поскольку не надо проверять результаты методов на ошибочность. Есть только один вариант выполнения запроса, который приводит к успеху: приложение получило валидные данные, сущность пользователя загружена из базы данных, старый пароль совпал, поменяли пароль на новый и сохранили все в базе. Любой шаг в сторону от этого единого пути должен вызывать исключение. Юзер ввёл невалидные данные - исключение, этому пользователю нельзя выполнить это действие - исключение, упал сервер с базой данных - разумеется, тоже исключение. Проблемой Единого Верного Пути является то, что где-то нужно будет отделить ошибки клиента от ошибок сервера, поскольку ответы мы должны сгенерировать разные (помните про 400-ые и 500-ые коды ответов?) да и логироваться такие ошибки должны по-разному.

Сложно сказать какой из путей предпочтительнее. Когда приложение только-только обзавелось отдельным слоем Приложения, второй путь кажется более приятным. Код чище, в любом приватном методе сервисного класса если что-то не понравилось можно просто выбросить исключение и оно сразу дойдёт до адресата. Однако если приложение будет расти дальше, например будет создан еще и Доменный слой, то это увлечение исключениями

может оказаться вредным. Некоторые из них, будучи выброшенными, но не пойманными на нужном уровне могут быть проинтерпретированы неверно на более высоком уровне. Количество try-catch блоков начнёт расти и код уже не будет таким чистым.

Laravel выбрасывает исключения для 404 ошибки, для ошибки доступа (код 403) да и вообще имеет класс **HttpException** в котором можно указать HTTP-код ошибки. Поэтому, в этой книге я тоже выберу второй вариант и буду генерировать исключения при любых проблемах.

Пишем код с исключениями:

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

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

        $user->password = password_hash($dto->newPassword);

        $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
    }
}
```

Даже на таком простом примере видно, что код Метода **UserService::changePassword** стал намного чище. Любой шаг в сторону от основной ветки выполнения вызывает исключение, которое ловится в контроллере. Eloquent тоже имеет методы для работы в таком стиле: **findOrFail()**, **firstOrFail()** и кое-какие другие ***OrFail()** методы. Правда этот код все ещё не без проблем:

1. **Exception::getMessage()** не самое лучшее сообщение для того, чтобы показывать пользователю. Сообщение “Old password is not valid” ещё неплохо, но, например, “Server Has Gone Away (error 2006)” точно нет.
2. Любые серверные ошибки должны быть записаны в лог. Мелкие приложения используют лог-файлы. Когда приложение становится популярным, исключения могут происходить каждую секунду. Некоторые исключения сигнализируют о проблемах в коде и должны быть исправлены немедленно. Некоторые исключения являются нормой: интернет не идеален, запросы в самые стабильные API один раз из миллиона могут заканчиваться неудачей. Однако если частота таких ошибок резко возрастает(перестало работать какое-то внешнее API), то разработчики должны среагировать тоже. В таких случаях, когда контроль за ошибками требует много внимания, лучше использовать специализированные сервисы, которые позволят группировать исключения и работать с ними намного удобнее. Если интересно, можете просто погуглить “error monitoring services” и найдёте

несколько таких сервисов. Большие компании строят свои специализированные решения для записи и анализа логов со всех своих серверов (часто на основе популярного на момент написания книги стэка ELK: Elastic, LogStash, Kibana). Некоторые компании не логируют ошибки клиента. Некоторые логируют, но в отдельных хранилищах. В любом случае, для любого приложения необходимо четко разделять ошибки сервера и клиента.

Базовый класс исключения

Первый шаг - создать базовый класс для всех исключений бизнес-логики таких, как “Старый пароль неверен”. В PHP есть класс **DomainException**, который мог бы быть использован с этой целью, но он уже используется в других местах, например в сторонних библиотеках и это может привести к путанице. Проще создать свой класс, скажем **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;
```

```
    }
}

// Теперь ошибка верификации старого пароля вызовет исключение

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) {
            // вернуть ошибочный ответ
            // с одним из 400-ых кодов
            // с $e->getUserMessage();
        } catch(\Throwable $e) {
            // залогировать ошибку

            // вернуть ошибочный ответ (с кодом 500)
            // с текстом "Houston, we have a problem"
            // Не возвращая реальный текст ошибки
        }

        // вернуть успешный ответ
    }
}
```

Этот код ловит **BusinessException** и показывает его сообщение пользователю. Другие исключения покажут некое “Внутренняя ошибка, мы работаем над этим” и исключение будет отправлено в лог. Код работает корректно, но секция **catch** будет повторена

один в один в каждом методе каждого контроллера. Стоит вынести логику обработки исключений на более высокий уровень.

Глобальный обработчик

В Laravel (как и почти во всех фреймворках) есть глобальный обработчик исключений и, как ни странно, здесь весьма удобно обрабатывать почти все исключения нашего приложения. В новых версиях Laravel он устроен по-другому. Я же рассмотрю класс **app/Exceptions/Handler.php** из старых версий. Класс **Handler** реализует две очень близкие ответственности: логирование исключений и сообщение пользователям о них.

```
namespace App\Exceptions;

class Handler extends ExceptionHandler
{
    protected $dontReport = [
        // Это означает, что BusinessException
        // не будет логироваться
        // но будет показан пользователю
        BusinessException::class,
    ];

    public function report(Exception $e)
    {
        if ($this->shouldReport($e))
        {
            // Это отличное место для
            // интеграции сторонних сервисов
            // для мониторинга ошибок
        }

        // это залогирует исключение
```

```
        // по умолчанию в файл laravel.log
        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())]);
            }
        }

        // Стандартный показ ошибки
        // такой как страница 404
        // или страница "Oops" для 500 ошибок
        return parent::render($request, $e);
    }
}
```

Простой пример глобального обработчика. Метод **report** может быть использован для дополнительного логирования. Вся **catch** секция из контроллера переехала в метод **render**. Здесь все ошиб-

ки логики будут отловлены и будут сгенерированы правильные сообщения для пользователя. Посмотрите на контроллер:

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

        // возвращаем успешный ответ
    }
}
```

Прекрасно. Бизнес-логика уехала из контроллера в сервисный класс. Валидация в Request-объект. Обработка исключений в глобальный обработчик. Контроллеру осталось лишь контролировать процесс на самом высоком уровне. Наконец-то, его работа соответствует названию!

Проверяемые и непроверяемые исключения

Закройте глаза. Сейчас я буду вещать о высоких материях, которые в конце концов окажутся бесполезными. Представьте себе берег моря и метод **UserService::changePassword**. Подумайте какие ошибки там могут возникнуть?

- **IlluminateDatabaseEloquentModelNotFoundException** если пользователя с таким **id** не существует
- **IlluminateDatabaseQueryException** если запрос в базу данных не может быть выполнен
- **AppExceptionsBusinessException** если старый пароль неверен

- **TypeError** если где-то глубоко внутри кода функция **foo(SomeClass $x)** получит параметр **$x** с другим типом
- **Error** если **$var->method()** будет вызван, когда переменная **$var** == **null**
- еще много других исключений

С точки зрения вызывающего этот метод, некоторые из этих ошибок, такие как **Error**, **TypeError**, **QueryException**, абсолютно вне контекста. Какой-нибудь HTTP-контроллер вообще не знает, что с этими ошибками делать. Единственное, что он может, это показать пользователю сообщение “Произошло что-то плохое и я не знаю, что с этим делать”. Но некоторые из них имеют смысл для него. **BusinessException** говорит о том, что что-то не так с логикой и там есть сообщения прямо для пользователя и контроллер точно знает, что с этим исключением делать. То же самое можно сказать про **ModelNotFoundException**. Контроллер может показать 404 ошибку на это. Да, мы вынесли все это из контроллеров в глобальный обработчик, но это не важно. Итак, два типа ошибок:

1. Ошибки, которые понятны вызывающему коду и могут быть эффективно обработаны там
2. Другие ошибки

Первые ошибки хорошо бы обработать там, где этот метод вызывается, а вторые можно и пробросить выше. Запомним это и взглянем на язык Java.

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

Этот код даже не скомпилируется. Сообщение компилятора: “Error:(5, 9) java: unreported exception java.lang.Exception; must be caught or declared to be thrown” Есть два способа исправить это. Поймать его:

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

Или описать исключение в сигнатуре метода:

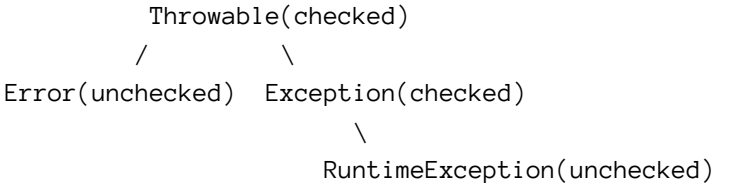
```
public class Foo
{
    public void bar() throws Exception
    {
        throw new Exception("test");
    }
}
```

В этом случае каждый код, вызывающий метод **bar** будет вынужден что-то делать с этим исключением:

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

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

Разумеется, работать так с **каждым** исключение будет той еще пыткой. В Java исключения делятся на два типа: 1. **проверяемые**(checked) исключения, которые обязаны быть пойманы или объявлены в сигнатуре 2. **непроверяемые**(unchecked), которые могут быть выброшены без всяких дополнительных условий.

Взглянем на корень дерева классов исключений в Java (PHP, начиная с седьмой версии, имеет такое же):



Throwable, **Exception** и все их наследники - проверяемые исключения. Кроме **Error**, **RuntimeException** и всех их наследников. Их можно выбросить везде и ничего за это не будет.

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

Что сигнатура метода **getCanonicalPath** говорит разработчику? Там нет никаких параметров, возвращает строку, может выбросить исключение **IOException**, а также любое непроверяемое исключение. Возвращаясь к двум типам ошибок:

1. Ошибки, которые понятны вызывающему коду и могут быть эффективно обработаны там
2. Другие ошибки

Проверяемые исключения созданы для ошибок первого типа. Непроверяемые - для второго. Вызывающий код может эффективно обработать проверяемое исключение, и эта строгость обязывает его сделать это. Все это приводит к более корректной работе с ошибками.

Хорошо, в Java это есть, в PHP - нет. Но IDE, которое я использую, PhpStorm, имитирует поведение Java.

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

PhpStorm подсветит ‘throw new Exception();’ с предупреждением: ‘Unhandled Exception’. Есть два пути избавиться от этого:

1. Поймать исключение
2. Описать его в тэге @throws phpDoc-комментария метода:

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

Список непроверяемых классов конфигурируется. По умолчанию он выглядит так: **Error**, **RuntimeException** и **LogicException**. Их можно выбрасывать не опасаясь предупреждений.

Со всей этой информацией можно попробовать спроектировать структуру классов исключения для приложения. Я бы хотел информировать код, вызывающий **UserService::changePassword** про ошибки:

1. **ModelNotFoundException**, когда пользователь с таким **id** не найден
2. **BusinessException**, эта ошибка содержит сообщение, предназначенное для пользователя и может быть обработана сразу. Все остальные ошибки могут быть обработаны позже. Итак, в идеальном мире:

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

class BusinessException extends \Exception
{...}

final class UserService
{
    /**
     * @param ChangeUserPasswordDto $command
     * @throws ModelNotFoundException
     * @throws BusinessException
     */
    public function changePassword(
        ChangeUserPasswordDto $command): void
    {...}
}
```

Но мы уже вынесли всю логику обработки ошибок в глобальный обработчик, поэтому придется копировать все эти **@throws** тэги в методе контроллера:

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

        // возвращаем успешный ответ
```

```
    }
}
```

Не очень удобно. Даже если учесть, что PhpStorm умеет генерировать все эти тэги автоматически. Возвращаясь к нашему неидеальному миру: Класс **ModelNotFoundException** в Laravel уже отнаследован от **RuntimeException**. Соответственно, он непроверяемый по умолчанию. Это имеет смысл, поскольку глубоко внутри собственного обработчика ошибок Laravel обрабатывает эти исключения сам. Поэтому, в нашем текущем положении, стоит тоже пойти на такую сделку с совестью:

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

и забыть про тэги **@throws** держа в голове то, что все исключения **BusinessException** будут обработаны в глобальном обработчике.

Это одна из главных причин почему новые языки не имеют такую фичу с проверяемыми исключениями и большинство Java-разработчиков не любят их. Другая причина: некоторые библиотеки просто пишут “throws Exception” в своих методах. “throws Exception” вообще не дает никакой полезной информации. Это просто заставляет клиентский код повторять этот бесполезный “throws Exception” в своей сигнатуре.

Я вернусь к исключениям в главе про Доменный слой, когда этот подход с непроверяемыми исключениями станет не очень удобным.

Пара слов в конце главы

Функция или метод, возвращающие более одного типа, могущие вернуть **null** или возвращающие булевое значение (хорошо все

прошло или нет), делают вызывающий код грязным. Возвращенное значение нужно будет проверять сразу после вызова. Код с исключениями выглядит намного чище:

```
// Без исключений
$user = User::find($command->getUserId());
if($user === null) {
    // обрабатываем ошибку
}

$user->doSomething();


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

С другой стороны, использование объектов как **FunctionResult** даёт разработчикам больший контроль над исполнением. Например, **findOrFail** вызванное в неправильном месте в неправильное время заставит приложение показать пользователю 404ю ошибку вместо корректного сообщения об ошибке. С исключениями надо всегда быть настороже.

7. Валидация

> “...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...”

Валидация связанная с базой данных

Как всегда, главу начнём с примера кода с практиками, накопленными в предыдущих главах. Создание статьи:

```
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
{
```

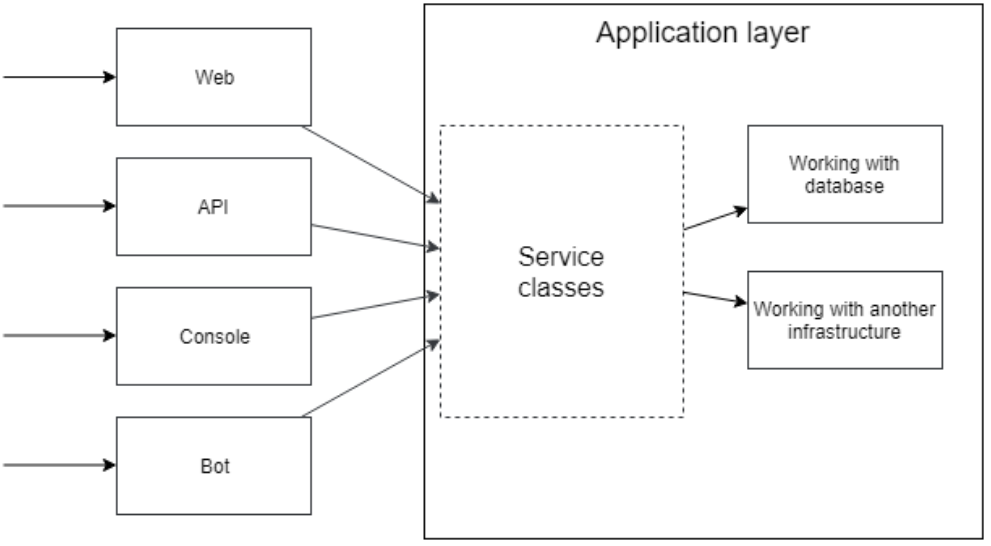
```
    public function create(CreatePostDto $dto)
    {
        $post = new Post();
        $post->category_id = $dto->categoryId;
        $post->title = $dto->title;
        $post->body = $dto->body;

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

Я, краткости ради, вернул валидацию обратно в контроллер. Одной из проверок является существование нужной категории в базе данных. Давайте представим, что в будущем в модель категорий будет добавлен трейт **SoftDeletes**. Этот функционал будет помечать строки в базе данных как удаленные вместо того, чтобы физически их удалять, а также не включать строки, помеченные как удаленные в результаты запросов. Всё приложение продолжит работать, не заметив этого изменения. Кроме этой валидации. Она позволит создать статью с удаленной категорией, нарушив тем самым консистентность данных. Исправим это:

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

Была добавлена проверка на пометку на удаление. Могу ещё добавить, что таких правок может понадобиться много, ведь статьи не только создаются. Любое другое изменение может опять сломать нашу “умную” валидацию. Например, пометка “archived” для категорий, которая позволит им оставаться на сайте, но не позволяет добавлять новые статьи в них. Мы не делали никаких

изменений в форме добавления статьи, и вообще во всей HTTP части приложения. Изменения касались либо бизнес-логики (архивные категории), либо логики хранения данных (Soft delete), однако менять приходится классы HTTP-запросов с их валидацией. Это ещё один пример высокой связанности (high coupling). Не так давно мы решили вынести всю работу с базой данных в Слой Приложения, но по старой привычке всё ещё лезем в базу напрямую из валидации, игнорируя все абстракции, которые строятся с помощью Eloquent или Слоя Приложения.



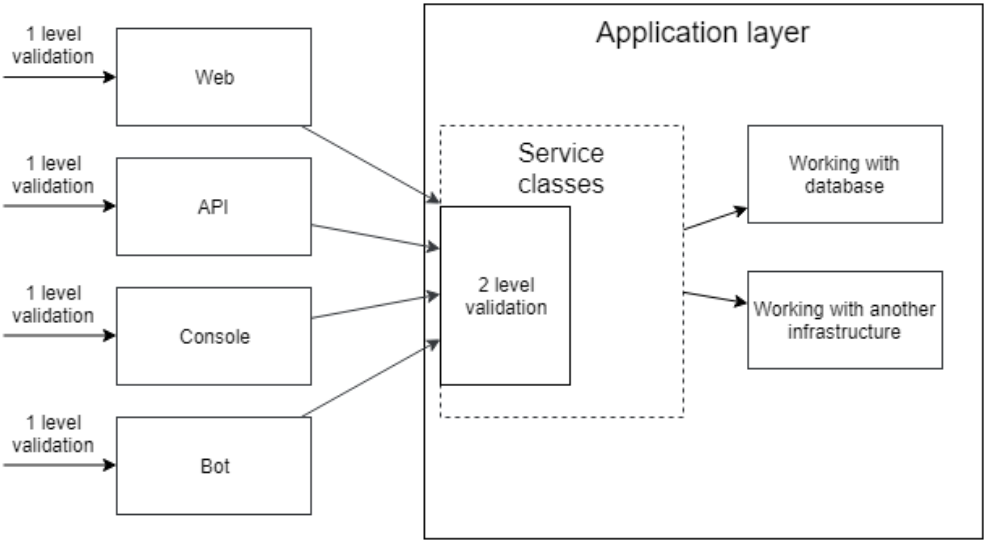
Надо разделить валидацию. В валидации HTTP слоя нам просто необходимо убедиться, что пользователь не ошибся при вводе данных:

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

class PostService
{
    public function create(CreatePostDto $dto)
    {
        $category = Category::find($dto->categoryId);

        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->title;
        $post->body = $dto->body;

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

Валидацию же, затрагивающую бизнес-логику или базу данных, необходимо проводить в более правильных местах. Теперь код работает более стабильно: валидация в контроллерах или **FormRequest** не меняется от случайных изменений в других слоях. Метод **PostService::create** не доверяет такие важные проверки вызывающему коду и валидирует всё сам. В качестве бонуса, приложение теперь имеет намного более понятные

тексты ошибок.

Два уровня валидации



В прошлом примере валидация была разделена на две части и я сказал, что метод **PostService::create** не доверяет сложную валидацию вызывающему коду, но он всё ещё доверяет ему в простом:

```
$post->title = $dto->title;
```

Здесь мы исходим из того, что заголовок у нас будет непустой, однако на 100 процентов уверенности нет. Да, сейчас оно проверяется правилом ‘required’ при валидации, но это далеко, где-то в контроллерах или ещё дальше. Метод **PostService::create** может быть вызван из другого кода, и там эта проверка может быть забыта. Давайте рассмотрим пример с регистрацией пользователя (он удобнее):

```
readonly final class RegisterUserDto
{
    public function __construct(
        public string $name,
        public string $email,
        public DateTime $birthDate,
    ) {}
}

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

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

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

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

После начала использования DTO мы вынуждены забыть про то, что данные в web запросе были отвалидированы. Любой может написать такой код:

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

Никто не может поручиться, что в **name** будет лежать непустая

строка, а в **email** строка с верным email адресом. Что делать? Можно дублировать валидацию в сервисном классе:

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

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

        //...
    }
}
```

Или сделать такую же валидацию в конструкторе DTO класса, но в приложении будет куча мест, где нужно будет получать email, имя или подобные данные. Много кода будет дублироваться. Я могу предложить два варианта.

Валидация аннотациями

Проект **Symfony** содержит отличный компонент для валидации аннотациями - **symfony/validator**. Перепишем наш **RegisterUserDto**:

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

readonly class RegisterUserDto
{
    public function __construct(
        #[Assert\NotBlank]
        private string $name;

        #[Assert\NotBlank]
        #[Assert\Email]
        private string $email;

        #[Assert\NotNull]
        private DateTime $birthDate;
    ) {}
}
```

Просим валидатор в сервисном классе и используем его для проверки DTO:

```
class UserService
{
    public function __construct(
        private ValidatorInterface $validator
    ) {}

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

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

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

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

        $user = new User();
        $user->name = $dto->name;
        $user->email = $dto->email;
        $user->birthDate = $dto->birthDate;

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

Правила валидации описываются аннотациями. Метод **ValidatorInterface::validate** возвращает список нарушений правил валидации. Если он пуст - всё хорошо. Если нет, выбрасываем исключение валидации - **ValidationException**. Используя эту явную валидацию, в Слое Приложения можно быть уверенным в валидности данных. Также, в качестве бонуса, можно удалить валидацию в слое Web, API и т.д, поскольку все данные уже проверяются глубже. Отличная идея, но с этим есть некоторые проблемы.

Проблема данных Http запроса

В первую очередь, данные, которые передаются от пользователей в HTTP-запросе, не всегда равны данным, передаваемым в Слой Приложения. Когда пользователь меняет свой пароль, приложение запрашивает старый пароль, новый пароль и повторить новый пароль. Валидация Web-слоя должна проверить поля нового пароля на совпадение, а Слою Приложения эти данные просто не нужны, он получит только значения старого и нового пароля.

Другой пример: одно из значений, передаваемых в Слой Приложения заполняется email-адресом текущего пользователя. Если

этот email окажется пустым, то пользователь может увидеть сообщение “Формат email неверный”, при том, что он даже не вводил никакого email! Поэтому, делать валидацию пользовательского ввода в Слое Приложения - не самая лучшая идея.

Проблема сложных структур данных

Представьте некий DTO создания заказа такси - **CreateTaxiOrderDto**. Это будет авиа-такси, поэтому заказы могут быть из одной страны в другую. Там будут поля **fromHouse**, **fromStreet**, **fromCity**, **fromState**, **fromCountry**, **toHouse**, **toStreet**, **toCity**,... Огромный DTO с кучей полей, дублирующих друг-друга, зависящих друг от друга. Номер дома не имеет никакого смысла без имени улицы. Имя улицы, без города и страны. Валидация подобных данных будет сложной и регулярно дублируемой в разных DTO-объектах.

Value objects

Решение этой проблемы лежит прямо в **RegisterUserDto**. Мы не храним отдельно **$birthDay**, **$birthMonth** и **$birthYear**. Не валидируем их каждый раз. Мы просто храним объект **DateTime**! Он всегда хранит корректную дату и время. Сравнивая даты, мы никогда не сравниваем их года, месяцы и дни. Там есть метод **diff()** для сравнений дат. Этот класс содержит все знания о датах внутри себя, избавляя нас от необходимости дублировать логику работы с ними везде. Можно попробовать сделать что-то похожее и с другими данными:

```php
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 createFromString(
        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 createFromString(
        string $name)
    {
        return new static($name);
    }

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

readonly final class RegisterUserDto
{
    public function __construct(
        public UserName $name,
        public Email $email,
        public DateTime $birthDate,
    ) {}
}
```

Да, создавать класс для каждого возможного типа вводимых данных - это не то, о чем мечтает каждый программист. Но это естественный путь декомпозиции приложения. Вместо того, чтобы использовать строки и всегда сомневаться, лежит ли в них нужное значение, эти классы позволяют всегда иметь корректные значения, как с DateTime. Этот шаблон называется **Объект-значение(Value Object** или **VO**). В поле **email** больше не лежит просто строка. Поле это теперь типа **Email**, который без сомнения

можно использовать везде, где нужны email-адреса. **UserService** может без страха использовать эти значения:

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

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

Для того чтобы полям Eloquent-сущности можно было присваивать такие VO как **Email**, необходимо реализовать их преобразование через механизм casting. Это влечет за собой дополнительные затраты и необходимо еще раз задуматься “а стоит ли данный проект таких затрат?”. Для многих проектов страхи, описанные выше, не так значительны и вполне можно обойтись без такого значительного усложнения логики. Однако, есть проекты, в которых цена ошибки будет слишком велика и такие усилия по защите целостности данных будут вполне оправданы.

Объект-значение как композиция других значений

Объекты-значения **Email** и **UserName** - это просто оболочки для строк, но шаблон **Объект-значение** - более широкое понятие. Географическая координата может быть описана двумя float значениями: долгота и широта. Обычно, мало кому интересна долго-

та, без знания широты. Создав объект **GeoPoint**, можно во всем приложении работать с ним.

```
readonly final class GeoPoint
{
    public function __construct(
        public float $latitude,
        public float $longitude,
    ) {}

    public function isEqual(GeoPoint $other): bool
    {
        // просто примера ради
        return $this->getDistance($other)->getMeters() < 10;
    }

    public function getDistance(GeoPoint $other): Distance
    {
        // Вычисление расстояния между $this и $other
    }
}

final class City
{
    //...
    private GeoPoint $centerPoint;

    public function getDistance(City $other): Distance
    {
        return $this->centerPoint
                    ->getDistance($other->centerPoint);
    }
}
```

Примером того, как знание о координатах инкапсулировано в классе **GeoPoint**, является метод **getDistance** класса **City**. Для вы-

численния дистанции между городами просто используется расстояние между двумя центральными точками городов.

Другие примеры объектов-значений:

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

Вы заметили, что в прошлом примере я пытаюсь не использовать примитивные типы, такие как строки и числа? Метод **getDistance()** возвращает объект **Distance**, а не int или float. Класс **Distance** может иметь методы **getMeters()**: float или **getMiles()**: float. А также **Distance::isEqual**(**Distance $other**) для сравнения двух расстояний. Это тоже объект-значение! Для многих проектов такая детализация излишня, и метод **GeoPoint::getDistance()**: возвращающий число с плавающей запятой расстояния в метрах более, чем достаточен. Я лишь хотел показать пример того, что я называю “мышлением объектами”. Мы ещё вернемся к объектам-значениям позднее в этой книге. Вероятно, вы понимаете, что этот шаблон слишком мощный, чтобы использоваться только как поле в DTO.

Объекты-значения и валидация

```
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
    }
}
```

В этом коде можно легко найти дубликаты. Значение email-адреса сначала валидируется с помощью Laravel валидации, а потом в конструкторе класса **Email**:

```
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;
    }
```

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

Идея удаления кода Laravel-валидации выглядит интересной. Можно удалить вызов **$this->validate()** и просто ловить исключение **InvalidArgumentException** в глобальном обработчике ошибок. Но, как я уже писал, данные HTTP-запроса не всегда равны данным, передаваемым в Слой Приложения, да и исключение **InvalidArgumentException** может быть выброшено во многих других ситуациях. Опять может повториться ситуация, когда пользователь видит ошибки про данные, которые он не вводил.

Если вы помните, PhpStorm по умолчанию имеет 3 класса непроверяемых исключений: **Error**, **RuntimeException** и **LogicException**:

- **Error** означает ошибку языка PHP, например **TypeError**, **ParseError**, и т.д.
- **RuntimeException** означает ошибку времени выполнения, не зависящую от нашего кода, например проблемы с соединением с базой данных.
- **InvalidArgumentException** наследует от **LogicException**. Описание **LogicException** в документации PHP: “Исключение означает ошибку в логике приложения. Эти ошибки должны напрямую вести к исправлению кода.”. Поэтому, если код написан верно, он никогда не должен выбрасывать **LogicException**.

Это означает, что проверки в конструкторах объектов-значений нужны только для того, чтобы убедиться, что данные были проверены ранее, например с помощью вызова метода ->validate() и стандартного валидатора Laravel. Они не должны быть использованы в качестве валидации данных, введенных пользователем. Это валидация кода нашего приложения.

Пара слов в конце главы

Вынесение логики в Слой Приложения ведет к некоторым проблемам с валидацией данных. Мы не можем напрямую использовать объекты **FormRequest** и приходится использовать некие объекты передачи данных (DTO), пусть даже это будут простые массивы. Если Слой Приложения всегда получает ровно те данные, которые ввел пользователь, то вся валидация может быть перенесена туда и использовано решение с пакетом **symfony/validator** или другим. Но это будет опасно и не очень удобно, если идет работа со сложными структурами данных, такими как адреса или точки координат, например.

Валидация может быть оставлена в Web, API и других частях кода, а Слой Приложения будет просто доверять переданным ему данным. По моему опыту это работает только в маленьких проектах. Большие проекты, над которыми работают команды разработчиков, постоянно будут сталкиваться с проблемами невалидных данных, которые будут вести к неправильным значениям в базе данных или к выбросу неверных исключений.

Шаблон объект-значение требует некоторого дополнительного кодинга и “мышления объектами” от программистов, но это наиболее безопасный и естественный способ представлять данные, имеющие какой-то дополнительный смысл, т.е. “не просто строка, а email”. Как всегда, это выбор между краткосрочной и долгосрочной производительностью.

8. События

Действие Слоя Приложения всегда содержит главную часть, где выполняется основная логика действия, а также некоторые дополнительные действия. Регистрация пользователя состоит из, собственно, создания сущности пользователя, а также посылки соответствующего письма. Обновление текста статьи содержит обновление значения **$post->text** с сохранением сущности, а также, например вызов **Cache::forget** для инвалидации кеша. Псевдо-реальный пример: сайт с опросами. Создание сущности опроса:

```
final class SurveyService
{
    public function __construct(
        private ProfanityFilter $profanityFilter
        /*, ... другие зависимости */
    ) {}

    public function create(SurveyCreateDto $request)
    {
        $survey = new Survey();
        $survey->question = $this->profanityFilter->filter(
            $request->getQuestion());
        //...
        $survey->save();

        foreach($request->getOptions() as $optionText)
        {
            $option = new SurveyOption();
            $option->survey_id = $survey->id;
            $option->text =
```

```
                    $this->profanityFilter->filter($optionText);
                $option->save();
            }

            // Вызов генерации sitemap

            // Оповестить внешнее API о новом опросе
        }
    }
```

Это обычное создание опроса с вариантами ответа, с фильтрацией мата во всех текстах и некоторыми пост-действиями. Объект опроса непростой. Он абсолютно бесполезен без вариантов ответа. Мы должны позаботиться о его консистентности. Этот маленький промежуток времени, когда мы уже создали объект опроса и добавили его в базу данных, но еще не создали его варианты ответа, очень опасен!

Database transactions

Первая проблема - база данных. В это время может произойти некоторая ошибка (с соединением с базой данных, да даже просто в функции проверки на мат) и объект опроса будет в базе данных, но без вариантов ответа. Все движки баз данных, которые созданы для хранения важных данных, имеют механизм транзакций. Они гарантируют консистентность внутри транзакции. Все запросы, обернутые в транзакцию, либо будут выполнены полностью, либо, при возникновении ошибки во время выполнения, ни один из них. Выглядит как решение:

```
final class SurveyService
{
    public function __construct(
        ..., private DatabaseConnection $connection
    ) {}

    public function create(SurveyCreateDto $request)
    {
        $this->connection->transaction(function() use ($request) {
            $survey = new Survey();
            $survey->question = $this->profanityFilter->filter(
                    $request->getQuestion());
            //...
            $survey->save();

            foreach($request->getOptions() as $optionText) {
                $option = new SurveyOption();
                $option->survey_id = $survey->id;
                $option->text =
                    $this->profanityFilter->filter($optionText);
                $option->save();
            }

            // Вызов генерации sitemap

            // Оповестить внешнее API о новом опросе
        });
    }
}
```

Отлично, наши данные теперь консистентны, но эта магия транзакций даётся базе данных не бесплатно. Когда мы выполняем запросы в транзакции, база данных вынуждена хранить две копии данных: для успешного или неуспешного результатов. Для проектов с нагрузкой, которые могут выполнять сотни одновременных транзакций, длящихся долго, это может сильно сказаться на

производительности. Для проектов с небольшой нагрузкой это не настолько важно, но все равно стоит приобрести привычку выполнять транзакции как можно быстрее. Проверка на мат может требовать запроса на специальное API, которое может занять страшно много времени. Попробуем вынести из транзакции все, что возможно:

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

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

            foreach($filteredRequest->getOptions()
                                as $optionText) {
                $option = new SurveyOption();
                $option->survey_id = $survey->id;
                $option->text = $optionText;
                $option->save();
            }
        });

    // Вызов генерации sitemap

    // Оповестить внешнее API о новом опросе
}

private function filterRequest(
    SurveyCreateDto $request): SurveyCreateDto
```

```
{
    // фильтрует тексты в запросе
    // и возвращает такое же DTO но с "чистыми" данными
}
}
```

Теперь внутри транзакции только легкие действия и она выполняется моментально. Так и должно быть.

Очереди

Вторая проблема - время выполнения запроса. Приложение должно отвечать как можно быстрее. Создание опроса содержит тяжелые действия, такие как генерация sitemap или вызовы внешних API. Обычное решение - отложить эти действия с помощью механизма очередей. Вместо того чтобы выполнять тяжелое действие в обработчике web-запроса, приложение может создать задачу для выполнения этого действия и положить его в очередь. Дальше из очереди его возьмет служба, расположенная на том же сервере. Или она будет на других серверах, созданных специально для обработки задач из очередей. Такие сервера называют воркер-серверами. Очередью может быть таблица в базе данных, список в Redis или специальный софт для очередей, например Kafka или RabbitMQ.

Laravel предоставляет несколько путей работы с очередями. Один из них: jobs. Как я говорил ранее, действие состоит из главной части и несколько второстепенных действий. Главное действие при создании сущности опроса не может быть выполнено без фильтрации мата, но все пост-действия можно отложить. Вообще, в некоторых ситуациях, когда действие занимает слишком много времени, можно его полностью отложить, сказав пользователю “Скоро выполним”, но это пока не наш случай.

```
final class SitemapGenerationJob implements ShouldQueue
{
    public function handle()
    {
        // Вызов генератора sitemap
    }
}

final class NotifyExternalApiJob implements ShouldQueue {}

use Illuminate\Contracts\Bus\Dispatcher;

final class SurveyService
{
    public function __construct(...,
        private Dispatcher $dispatcher
    ) {}

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

        $survey = new Survey();
        $this->connection->transaction(...);

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

Если класс содержит интерфейс **ShouldQueue**, то выполнение этой задачи будет отложено в очередь. Этот код выполняется достаточно быстро, но мне он все ещё не нравится. Таких постдействий может быть очень много и сервисный класс начинает

знать слишком много. Он выполняет каждое пост-действие, но с высокоуровневой точки зрения, действие создания опроса не должно знать про генерацию sitemap или взаимодействие с внешними API. В проектах с огромным количеством пост-действий, контролировать их становится очень непросто.

События

Вместо того чтобы напрямую вызывать каждое пост-действие, сервисный класс может просто информировать приложение о том, что что-то произошло. Приложение может реагировать на эти события, выполняя нужные пост-действия. В Laravel есть поддержка механизма событий:

```
final class SurveyCreated
{
    public function __construct(
        public readonly int $surveyId
    ) {}
}

use Illuminate\Contracts\Events\Dispatcher;

final class SurveyService
{
    public function __construct(...,
        private Dispatcher $dispatcher
    ) {}

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

        $survey = new Survey();
```

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

        $this->dispatcher->dispatch(new SurveyCreated($survey->id));
    }
}

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

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

Теперь Слой Приложения просто сообщает, что был создан новый опрос (событие **SurveyCreated**). Приложение имеет конфигурацию для реагирования на события. Классы слушателей событий (**Listener**) содержат реакцию на события. Интерфейс **ShouldQueue** работает точно также, сообщая о том, когда должно быть запущено выполнение этого слушателя: сразу же или отложено. События - очень мощная вещь, но здесь тоже есть ловушки.

Использование событий Eloquent

Laravel генерирует кучу событий. События системы кеширования: **CacheHit**, **CacheMissed**, и т.д.. События оповещений: **NotificationSent**, **NotificationFailed**, и т.д.. Eloquent тоже генерирует свои события. Пример из документации:

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

Событие **UserSaved** будет генерироваться каждый раз когда сущность пользователя будет сохранена в базу данных. Сохранена, означает любой update или insert запрос. Использование этих событий имеет множество недостатков.

UserSaved не самое удачное название для этого события. **UsersTableRowInsertedOrUpdated** более подходящее. Но и оно не всегда верное. Это событие не будет сгенерировано при массовых операциях со строками базы данных. Событие “Deleted” не будет вызвано, если строка в базе данных будет удалена с помощью механизма каскадного удаления в базе данных. Главная же проблема это то, что это события уровня инфраструктуры, события строчек базы данных, но они используются как бизнес-события, или события доменной области. Разницу легко осознать в примере с созданием сущности опроса:

```
final class SurveyService
{
public function create(SurveyCreateDto $request)
{
    //...
    $this->connection->transaction(function() use (...) {
        $survey = new Survey();
        $survey->question = $filteredRequest->getQuestion();
        //...
        $survey->save();

        foreach($filteredRequest->getOptions() as $optionText){
            $option = new SurveyOption();
            $option->survey_id = $survey->id;
            $option->text = $optionText;
            $option->save();
        }
    });
    //...
}
}
```

Вызов **$survey->save();** сгенерирует событие ‘saved’ для этой сущности. Первая проблема в том, что объект опроса еще не готов и неконсистентен. Он все еще не имеет вариантов ответа. В слушателе, который захочет отправить email с этим опросом, явно хочется иметь этот объект полностью, со всеми вариантами ответа. Для отложенных слушателей это не проблема, но на машинах разработчиков часто значение **QUEUE_DRIVER** - ‘sync’, поэтому все отложенные слушатели и задачи будут выполняться сразу и поведение будет разным. Я сильно рекомендую избегать кода, который правильно работает “иногда, в некоторой особой ситуации”, а иногда может преподнести неприятный сюрприз.

Вторая проблема - эти события вызываются прямо внутри транзакции. Выполнение слушателей сразу или даже отправка их в очередь делают транзакции более долгими и хрупкими. А самое

страшное, что событие вроде **SurveyCreated**, может быть вызвано, но дальше в транзакции будет ошибка и вся она будет откачена назад. Письмо же пользователю об опросе, который даже не создался, все равно будет отправлено. Я нашел пару пакетов для Laravel, которые ловят все эти события, хранят их временно, и выполняют только после того, как транзакция будет успешно завершена (гуглите “Laravel transactional events”). Да, они решают множество из этих проблем, но всё это выглядит так неестественно! Простая идея генерировать нормальное бизнес-событие **SurveyCreated** после успешной транзакции намного лучше.

Сущности как поля классов-событий

Я часто вижу как Eloquent-сущности используются напрямую в полях событий:

```
final class SurveyCreated
{
    public function __construct(
        public readonly Survey $survey
    ) {}
}

final class SurveyService
{
    public function create(SurveyCreateDto $request)
    {
        // ...
        $survey = new Survey();
        // ...
        $this->dispatcher->dispatch(new SurveyCreated($survey));
    }
}
```

```
final class SendSurveyCreatedEmailListener implements ShouldQueue
{
    public function handle(SurveyCreated $event)
    {
        // ...
        foreach($event->survey->options as $option)
        {...}
    }
}
```

Это простой пример слушателя, который использует значения **HasMany**-отношения. Этот код работает. Когда выполняется код **$event->survey->options** Eloquent делает запрос в базу данных и получает все варианты ответа. Другой пример:

```
final class SurveyOptionAdded
{
    public function __construct(
        public readonly Survey $survey
    ) {}
}

final class SurveyService
{
public function addOption(SurveyAddOptionDto $request)
{
    $survey = Survey::findOrFail($request->getSurveyId());

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

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

    $this->dispatcher->dispatch(new SurveyOptionAdded($survey));
}
```

```
}

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

А вот тут уже не все хорошо. Когда сервисный класс проверяет количество вариантов ответа, он получает свежую коллекцию текущих вариантов ответа данного опроса. Потом он добавляет новый вариант ответа, вызвав **$survey->options()->create(...)**; Дальше, слушатель, выполняя **$event->survey->options** получает старую версию вариантов ответа, без новосозданной. Это поведение Eloquent, который имеет два механизма работы с отношениями. Метод **options()** и псевдо-поле **options**, которое вроде бы и соответствует этому методу, но хранит свою версию данных. Поэтому, передавая сущность в события, разработчик должен озаботиться консистентностью значений в отношениях, например вызвав:

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

до передачи объекта в событие. Это все делает код приложения весьма хрупким. Его легко сломать неосторожной передачей объекта в событие. Намного проще просто передавать **id** сущности. Каждый слушатель всегда может получить свежую версию сущности, запросив её из базы данных по этому **id**.

Пара слов в конце главы

С развитием приложения, а особенно с ростом нагрузки на сервера, появляется естественное желание сократить время ответа сервера и время выполнения транзакций базы данных.

Все тяжелые действия лучше выполнить в другом потоке. Как правило, с помощью очередей. В этом нет какой-то серьезной сложности. Вполне достаточно организованно и скрупулезно подойти к данному вопросу. В сложном кейсе, когда количество действий и пост-действий становится большим, события и настройка их слушателей сильно помогает организовать все не превращая код приложения в хаос.

С транзакциями также нет какой-либо сложности, но неявная и скрытая логика, такая как события **Eloquent**-моделей, могут внести неразбериху в этот процесс. Явные события, вызываемые в контролируемых местах, намного более стабильная и управляемая стратегия. Событие **UserBanned** явно выражает, что произошло, в отличие от сильно более общего **UserSaved**.

9. Unit-тестирование

Первые шаги

Вы, вероятно, уже слышали про unit-тестирование. Оно довольно популярно сейчас. Я довольно часто общаюсь с разработчиками, которые утверждают, что не начинают писать код, пока не напишут тест для него. TDD-маньяки! Начинать писать unit-тесты довольно сложно, особенно если вы пишете, используя фреймворки такие, как Laravel. Unit-тесты одни из лучших индикаторов качества кода в проекте. Фреймворки пытаются сделать процесс добавления новых фич как можно более быстрым, позволяя срезать углы в некоторых местах, но высоко-связанный код обычная тому цена. Сущности железно связанные с базой данных, классы с большим количеством зависимостей, которые бывает трудно найти (Laravel фасады). В этой главе я постараюсь протестировать код Laravel приложения и показать главные трудности, но начнем с самого начала.

Чистая функция - это функция, результат которой зависит **только** от введенных данных. Она не меняет никакие внешние значения и просто вычисляет результат. Примеры:

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

Чистые функции очень простые и предсказуемые. Unit-тесты для них писать легко. Попробуем написать простую функцию (это может быть и методом класса) методом TDD:

```
function cutString(string $source, int $limit): string
{
    return ''; // начнем просто возвращая пустую строку
}

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));
    }
}
```

Я здесь использую PHPUnit для написания тестов. Название функции не очень удачное, но просто взглянув на тесты, можно понять что она делает. Тесты проверяют результат с помощью assertEquals. Unit-тесты могут служить документацией к коду, если они такие же простые и легко-читаемые.

Если я запущу эти тесты, то получу такой вывод:

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

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

Разумеется, ведь наша функция еще не написана. Время ее написать:

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

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

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

Вывод PHPUnit после этих правок:

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

Отлично! Класс unit-теста содержит список требований к функции:

- Для пустой строки результат тоже должен быть пуст.
- Для строк, которые короче лимита, должна вернуться эта строка без изменений.
- Для строк длиннее лимита, результатом должна стать строка, укороченная до этого лимита с тремя точками в конце.

Успешные тесты говорят о том, что код удовлетворяет требованиям. Но это не так! В коде небольшая ошибка и функция не работает как задумано, если длина строки совпадает с лимитом. Хорошая привычка: если найден баг, надо написать тест, который его воспроизведет и упадёт. Нам в любом случае нужно будет проверить исправлен ли этот баг и unit-тест хорошее место для этого. Новые тест-методы:

```
class CutStringTest extends \PHPUnit\Framework\TestCase
{
    // старые тесты

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

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

testBeyondTheLimit выполняется хорошо, а **testLimit** падает:

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

Исправление простое: поменять < на <=

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

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

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

Сразу же запускаем тесты:

```
OK (5 tests, 5 assertions)
```

Отлично. Проверка краевых значений (0, длина **$limit**, длина **$limit**+1, и т.д.) очень важная часть тестирования. Многие ошибки находятся именно в этих местах.

Когда я писал функцию **cutString**, я думал, что длина исходной строки мне понадобится дальше и сохранил её в переменную. Но оказалось, что дальше нам нужна только переменная **$limit**. Теперь я могу удалить эту переменную.

```
function cutString(string $source, int $limit): string
{
    if(strlen($source) <= $limit) {
        return $source;
    }

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

И опять: запускаем тесты! Я изменил код и мог что-то сломать при этом. Лучше это обнаружить как можно скорее и исправить. Эта привычка сильно повышает итоговую производительность.

С хорошо написанными тестами, почти любая ошибка будет поймана сразу и разработчик может исправить её пока тот код, который он поменял, у него всё еще в голове.

Я всё внимание обратил на главный функционал и забыл про пред-условия. Разумеется, параметр **$limit** в реальном проекте никогда не будет слишком маленький, но хороший дизайн функции предполагает проверку этого значения тоже:

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

    public function testLimitCondition()
    {
        $this->expectException(InvalidArgumentException::class);

        cutString('limit', 4);
    }
}

function cutString(string $source, int $limit): string
{
    if($limit < 5) {
        throw new InvalidArgumentException(
            'The limit is too low');
    }

    if(strlen($source) <= $limit) {
        return $source;
    }

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

Вызов **expectException** проверяет то, что исключение будет вы-

брошено. Если этого не произойдет, то тест будет признан упавшим.

Тестирование классов с состоянием

Чистые функции прекрасны, но в реальном мире слишком много вещей, которые нельзя описать исключительно ими. Объекты могут иметь **состояние**. Unit-тестирование классов с состоянием немного сложнее. Для таких тестов есть рекомендация делить код теста на три части: 1. **инициализация** объекта в нужном состоянии 2. **выполнение** тестируемого действия 3. **проверка** результата

Есть также шаблон AAA: Arrange, Act, Assert, который описывает те же три шага.

Начну с простого примера теста воображаемой сущности **Статья**, которая не является Eloquent моделью. Её можно создать только с непустым заголовком, а текст может быть пустым. Но опубликовать эту статью можно только, если её текст не пустой.

```
class Post
{
    public string $title;
    public string $body;
    public bool $published = false;

    public function __construct(
        $title, $body
    ) {
        if (empty($title)) {
            throw new InvalidArgumentException(
                'Title should not be empty');
```

```
        }

        $this->title = $title;
        $this->body = $body;
    }

    public function publish()
    {
        if (empty($this->body)) {
            throw new CantPublishException(
                'Cant publish post with empty body');
        }

        $this->published = true;
    }
}
```

Конструктор класса **Post** - чистая функция, поэтому тесты для нее подобны предыдущим:

```
class CreatePostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulCreate()
    {
        // инициализация и выполнение
        $post = new Post('title', '');

        // проверка
        $this->assertEquals('title', $post->title);
    }

    public function testEmptyTitle()
    {
        // проверка
        $this->expectException(InvalidArgumentException::class);
```

```
        // инициализация и выполнение
        new Post('', '');
    }
}
```

Однако, метод **publish** зависит от текущего состояния объекта и части тестов более ощутимы:

```
class PublishPostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulPublish()
    {
        // инициализация
        $post = new Post('title', 'body');

        // выполнение
        $post->publish();

        // проверка
        $this->assertTrue($post->published);
    }

    public function testPublishEmptyBody()
    {
        // инициализация
        $post = new Post('title', '');

        // проверка
        $this->expectException(CantPublishException::class);

        // выполнение
        $post->publish();
    }
}
```

При тестировании исключений **проверка**, которая обычно последняя, происходит до **выполнения**. Тестирование классов с

состоянием сложнее тестирования чистых функций, поскольку разработчик должен держать в голове состояние объекта и проверить все возможные варианты пар “состояние-изменение”.

Тестирование классов с зависимостями

Одной из важных особенностей unit-тестирования является тестирование в изоляции. Unit (класс, функция или другой модуль) должен быть изолирован от всего остального мира. Это будет гарантировать, что тест тестирует только этот модуль. Тест может упасть только по двум причинам: неправильный тест или неправильный код тестируемого модуля. Ни неправильно настроенная база данных, ни появившийся баг в какой-то из используемых библиотек не могут уронить unit-тесты. Тестирование в изоляции даёт нам эту простоту и быстродействие. Настоящие unit-тесты выполняются очень быстро, поскольку во время их выполнения не происходит никаких тяжелых операций, вроде запросов в базу данных, чтения файлов или вызовов API. Когда класс просит некоторые DI-зависимости, тест должен их ему предоставить.

Зависимости на реальные классы

В главе про внедрение зависимостей я писал про два типа возможных интерфейсов:

1. Есть интерфейс и несколько возможных реализаций.
2. Есть интерфейс и одна реализация.

Для второго случая я предлагал не создавать интерфейса, теперь же хочу проанализировать это. Какая зависимость может быть реализована только одним возможным способом? Все операции ввода/вывода, такие как вызовы API, операции с файлами или

запросы в базу данных, всегда могут иметь другие возможные реализации. С другим драйвером, декоратором и т.д. Иногда класс содержит некоторые большие вычисления и разработчик решает вынести эту логику в отдельный класс. Этот новый класс становится новой зависимостью. В этом случае трудно себе представить другой возможный вариант реализации этой зависимости и это прекрасный момент, чтобы поговорить про инкапсуляцию и почему unit-тестирование называется unit-тестированием, т.е. тестированием модулей, а не тестирование классов или функций.

Это пример описанного случая. Класс **TaxCalculator** был вынесен в свой класс из класса **OrderService**.

```
class OrderService
{
    public function __construct(
        private TaxCalculator $taxCalculator
    ) {}

    public function create(OrderCreateDto $orderCreateDto)
    {
        $order = new Order();
        //...
        $order->sum = ...;
        $order->taxSum = $this->taxCalculator
                            ->calculateTax($order);
        //...
    }
}
```

Но если мы взглянем на класс **OrderService**, то увидим, что **TaxCalculator** не выглядит его зависимостью. Он не выглядит как что-то внешнее, нужное **OrderService** для работы. Он выглядит как часть класса **OrderService**.

OrderService здесь является модулем, который содержит не только класс **OrderService**, но и класс **TaxCalculator**. Класс **TaxCalculator** должен быть внутренней зависимостью, а не внешней.

```
class OrderService
{
    private TaxCalculator $taxCalculator = new TaxCalculator();
    //...
}
```

Теперь всему остальному коду необязательно знать про **TaxCalculator**. Unit-тесты могут тестировать класс **OrderService** не заботясь о предоставлении ему объекта **TaxCalculator**. Если условия изменятся и **TaxCalculator** станет внешней зависимостью (разные алгоритмы подсчета налогов), то зависимость будет несложно сделать публичной, нужно будет просто поставить его как параметр в конструктор и поменять код тестов.

Модуль - весьма широкое понятие. В начале этой статьи модулем была маленькая функция, а иногда в модуле может содержаться несколько классов. Программные объекты внутри модуля должны быть сфокусированы на одной ответственности, другими словами, иметь сильную связность. Когда методы класса полностью независимы друг от друга, класс не является модулем. Каждый метод класса - это модуль в данном случае. Возможно, стоит вынести эти методы в отдельные классы, чтобы разработчики не просматривали кучу лишнего кода каждый раз?

Стабы и фейки

Обычно, зависимость - это интерфейс, который имеет несколько реализаций. Использование реальных реализаций этого интерфейса во время unit-тестирования - плохая идея, поскольку там могут проводиться те самые операции ввода-вывода, замедляющие тестирование и не дающие провести тестирование этого модуля в изоляции. Прогон unit-тестов должен быть быстр как молния, поскольку запускаться они будут часто и важно, чтобы разработчик запустив их не потерял фокус над кодом. Написал код - прогнал тесты, еще написал код - прогнал тесты. Быстрые тесты позволят ему оставаться более продуктивным, не позволяя отвлекаться. Решение в лоб задачи изоляции класса от зависимостей - создание отдельной реализации этого интерфейса, предназначенного просто для тестирования. Вернемся к предыдущему примеру и вообразим, что **TaxCalculator** стал зависимостью и это теперь интерфейс с некоей реализацией.

```
interface TaxCalculator
{
    public function calculateTax(Order $order): float;
}

class OrderService
{
    public function __construct(
        private TaxCalculator $taxCalculator
    ) {}

    public function create(OrderCreateDto $orderCreateDto)
    {
        $order = new Order();
        //...
        $order->sum = ...;
        $order->taxSum = $this->taxCalculator
                            ->calculateTax($order);
```

```
        //...
    }
}

class FakeTaxCalculator implements TaxCalculator
{
    public function calculateTax(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
    }
}
```

Работает! Такие классы называются фейками. Библиотеки для unit-тестирования могут создавать такие классы на лету. Тот же самый тест, но с использованием метода **createMock** библиотеки PHPUnit:

```php
class OrderServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreate()
    {
        $stub = $this->createMock(TaxCalculator::class);

        $stub->method('calculateTax')
            ->willReturn(0);

        $orderService = new OrderService($stub);

        $orderService->create(new OrderCreateDto(...));

        // some assertions
    }
}
```

Стабы удобны когда нужно быстро настроить простую реализацию, однако когда тестов с этой зависимостью становится много, вариант с фэйковым классом смотрится оптимальнее. Библиотеки могут создавать стабы не только для интерфейсов, но и для реальных классов, что может быть весьма удобно при работе с легаси-проектами или для ленивых разработчиков.

Моки

Иногда разработчик хочет протестировать вызваны ли методы стаба, сколько раз и какие параметры переданы были.

> Вообще, я не считаю идею тестирования вызовов методов зависимостей хорошей идеей. Unit-тест в этом случае начинает знать слишком многое о том, как этот класс работает. Как следствие, такие тесты очень легко ломаются. Небольшой рефакторинг и тесты падают. Если это случается слишком часто, команда может просто забыть

> про unit-тестирование. Это называется тестированием методом белого ящика. Тестирование методом черного ящика пытается тестировать только входные и выходные данные, не залезая внутрь. Разумеется, тестирование черным ящиком намного стабильнее.

Эти проверки могут быть реализованы в фейковом классе, но это будет весьма непросто и мало кто захочет делать это для каждой возможной зависимости. Библиотеки для тестирования могут создавать специальные мок-классы, которые позволяют легко проверять вызовы их методов.

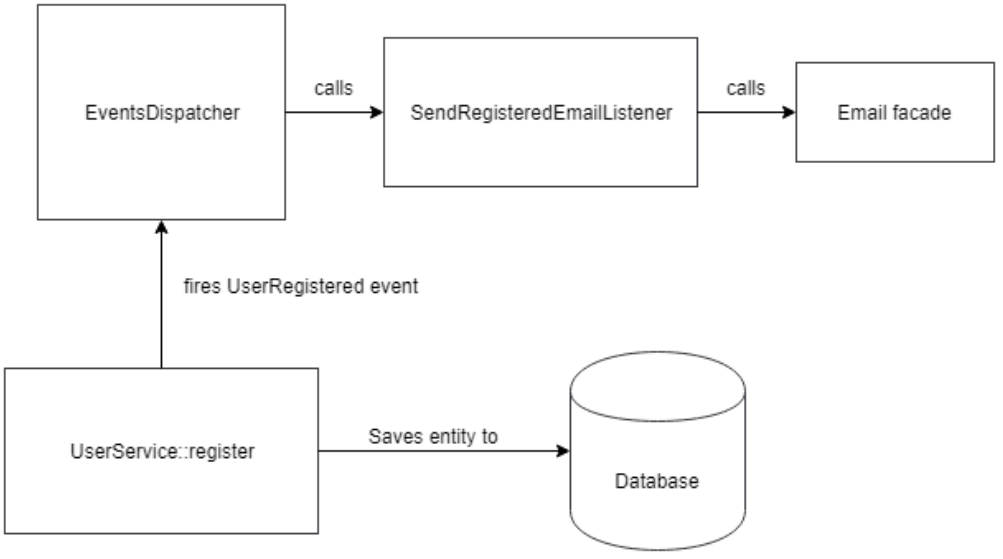
```
class OrderServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreate()
    {
        $stub = $this->createMock(TaxCalculator::class);

        // Конфигурация мок-класса
        $stub->expects($this->once())
            ->method('calculateTax')
            ->willReturn(0);

        $orderService = new OrderService($stub);

        $orderService->create(new OrderCreateDto(...));

        // некоторые проверки
    }
}
```

Теперь тест проверяет, что во время выполнения метода **OrderService::create** с переданными параметрами **TaxCalculator::calculateTax** был вызван ровно один раз. С мок-классами можно делать различные проверки на значения параметров и количество вызовов, настраивать

возвращаемые значения, выбрасывать исключения и т.д. Я не хочу фокусироваться на этом в данной книге. Фейки, стабы и моки имеют общее имя - test doubles, название для объектов, которые подставляются вместо реальных с целью тестирования. Они могут использоваться не только в unit-тестах, но в и интеграционных тестах.

Типы тестов ПО

Люди придумали множество способов тестировать приложения. Security testing для проверки приложений на различные уязвимости. Performance testing для проверки насколько хорошо приложение ведет себя при нагрузке. В этой главе мы сфокусируемся на проверке корректности работы приложений. Unit-тестирование уже было рассмотрено. **Интеграционное тестирование** проверяет совместную работу нескольких модулей. Пример: Попросить **UserService** зарегистрировать нового пользователя и проверить, что новая строка создана в базе данных, нужное событие (**UserRegistered**) было сгенерировано и соответствующий email был послан (ну или хотя бы фреймворк получил команду сделать это).



Функциональное тестирование (приёмочное или E2E - end to end) проверяет приложение на соответствие функциональным требованиям. Пример: Требование о создании некоей сущности. Тест открывает браузер, идёт на специфическую страницу, заполняет поля значениями, “нажимает” кнопку Создать и проверяет, что нужная сущность была создана, путем поиска её на определенной странице.

Тестирование в Laravel

Laravel предоставляет много инструментов для различного тестирования.

Инструменты Laravel для функционального тестирования

Инструменты для тестирования HTTP запросов, браузерного и консоли делают функциональное тестирование в Laravel весьма удобным, но мне не нравятся примеры из документации. Один из них, совсем немного измененный:

```
class ExampleTest extends TestCase
{
    public function testBasicExample()
    {
        $response = $this->postJson('/users', [
            'name' => 'Sally',
            'email' => 'sally@example.com'
        ]);

        $response
            ->assertOk()
            ->assertJson([
                'created' => true,
            ]);
    }
}
```

Этот тест просто проверяет, что запрос **POST /user** вернул успешный результат. Это не выглядит законченным тестом. Тест должен проверять, что пользователь реально создан. Но как? Первый ответ, приходящий в голову: просто сделать запрос в базу данных и проверить это. Опять пример из документации:

```
class ExampleTest extends TestCase
{
    public function testDatabase()
    {
        // Сделать запрос на создание пользователя

        $this->assertDatabaseHas('users', [
            'email' => 'sally@example.com'
        ]);
    }
}
```

Хорошо. Напишем другой тест подобным образом:

```
class PostsTest extends TestCase
{
    public function testDelete()
    {
        $response = $this->deleteJson('/posts/1');

        $response->assertOk();

        $this->assertDatabaseMissing('posts', [
            'id' => 1
        ]);
    }
}
```

А вот тут уже небольшая ловушка. Абсолютно такая же как и в главе про валидацию. Просто добавив трейт **SoftDeletes** в класс **Post**, мы уроним этот тест. Однако, приложение работает абсолютно также, выполняет те же самые требования и пользователи этой разницы не заметят. Функциональные тесты не должны падать в таких условиях. Тест, который делает запрос в приложение, а потом лезет в базу данных проверять результат, не является настоящим функциональным тестом. Он знает слишком

многое про то, как работает приложение, как оно хранит данные и какие таблицы для этого использует. Это еще один пример тестирования методом белого ящика.

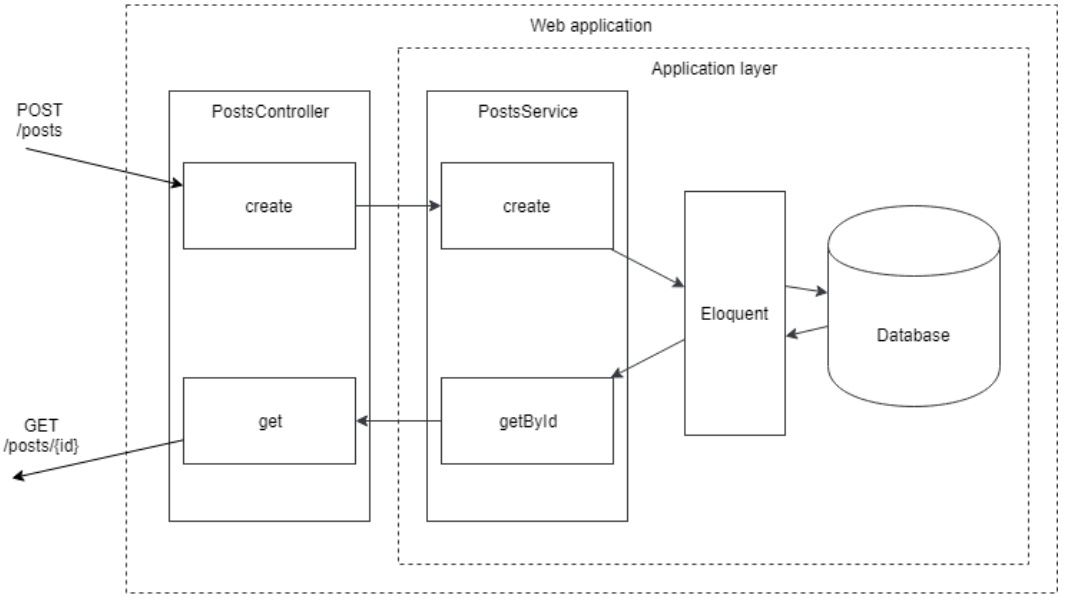
Как я уже говорил, функциональное тестирование проверяет, удовлетворяет ли приложение функциональным требованиям. Функциональное тестирование не про базу данных, оно о приложении в целом. Поэтому нормальные функциональные тесты не лезут внутрь приложения, они работают снаружи.

```
class PostsTest extends TestCase
{
    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()
    {
        // Здесь некоторая инициализация, чтобы создать
```

```
        // объект Post с id = $postId

        // Удостоверяемся, что этот объект есть
        $this->getJson('/api/posts/' . $postId)
            ->assertOk();

        // Удаляем его
        $this->jsonDelete('/posts/' . $postId)
            ->assertOk();

        // Проверяем, что больше в приложении его нет
        $this->getJson('/api/posts/' . $postId)
            ->assertStatus(404);
    }
}
```

Этому тесту абсолютно все равно как удален объект, с помощью ‘delete’ SQL запроса или с помощью Soft delete шаблона. Функциональный тест проверяет поведение приложения в целом. Ожидаемое поведение, если объект удален - он не возвращается по своему id и тест проверяет именно это.

Схема процессинга запросов “POST /posts/” и “GET /post/{id}”:



Что должен видеть функциональный тест:

Моки Laravel-фасадов

Laravel предоставляет удобную реализацию шаблона **Service Locator** - Laravel-фасады. Laravel не только предлагает их использовать, но и предоставляет инструменты для тестирования кода, который использует фасады. Давайте напишем один из предыдущих примеров с использованием Laravel-фасадов и протестируем этот код:

```
final class Survey extends Model
{
    public function options()
    {
        return $this->hasMany(SurveyOption::class);
    }
}

final class SurveyOption extends Model
{
}

final class SurveyCreated
{
    public function __construct(
        public readonly int $surveyId
    ) {}
}

final class SurveyCreateDto
{
    public function __construct(
        public readonly string $title,
        /** @var string[] */
        public readonly array $options,
    ) {}
}
```

```
final class SurveyService
{
    public function create(SurveyCreateDto $dto)
    {
        if(count($dto->options) < 2) {
            throw new BusinessException(
                "Please provide at least 2 options");
        }

        $survey = new Survey();

        \DB::transaction(function() use ($dto, $survey) {
            $survey->title = $dto->title;
            $survey->save();

            foreach ($dto->options as $option) {
                $survey->options()->create([
                    'text' => $option,
                ]);
            }
        });

        \Event::dispatch(new SurveyCreated($survey->id));
    }
}

class SurveyServiceTest extends TestCase
{
    public function testCreate()
    {
        \Event::fake();

        $postService = new SurveyService();
        $postService->create(new SurveyCreateDto(
            'test title',
```

```
            ['option1', 'option2']));

        \Event::assertDispatched(SurveyCreated::class);
    }
}
```

- Вызов **Event::fake()** трансформирует Laravel-фасад **Event** в мок-объект.
- Метод **SurveyService::create** создаёт опрос с вариантами ответа, сохраняет его в базу данных и генерирует событие **SurveyCreated**.
- Вызов **Event::assertDispatched** проверяет, что это событие было вызвано.

Я вижу несколько недостатков:

- Это не является unit-тестом. Фасад **Event** был заменен моком, но база данных - нет. Реальные строчки будут добавлены в базу данных. Для того чтобы сделать этот тест более чистым, в класс теста обычно добавляют трейт **RefreshDatabase**, который создаёт базу данных заново каждый раз. Это очень медленно. Один такой тест может быть выполнен за разумное время, но сотни таких займут уже несколько минут и никто не будет их выполнять после каждой мелкой правки.
- Тесты проверяют только генерацию событий. Для того чтобы проверить создание записей в базе данных нужно использовать вызовы методов, таких как **assertDatabaseHas** или что-то вроде **SurveyService::getById**, которое делает этот тест неким функциональным тестом к Слою Приложения, поскольку он просит Слой Приложения что-то сделать и проверяет результат тоже вызвав его.
- Зависимости класса **SurveyService** не описаны явно. Фасад **Event** вызывается где-то внутри. Чтобы понять, что конкретно он требует для своей работы, нужно просмотреть весь его

> код. Это делает написание тестов для него весьма неудобным. Хуже всего, если в класс будет добавлена новая зависимость с помощью laravel-фасада. Тесты будут продолжать работать как ни в чем не бывало, но не с моком, а с реальной реализацией этого фасада: реальные вызовы API и т.д. Я слышал несколько реальных историй, когда разработчики запускали тесты и это приводило к тысячам реальных денежных переводов! Ноги у такого растут именно из-за таких вот случаев, когда неожиданно в тесты попадет реальная реализация.

Я называю это “форсированным интеграционным тестированием”. Разработчик хочет написать unit-тест, но код обладает такой высокой связанностью, так крепко сцеплен с фреймворком, что этого просто не получается. Попробуем это сделать!

Unit-тестирование Слоя Приложения

Отсоединяем код от laravel-фасадов

Для того чтобы протестировать метод **SurveyService::create** в изоляции, нужно убрать использование Laravel-фасадов и базы данных (Eloquent). Первая часть несложная, у нас есть Внедрение Зависимостей.

- Фасад **Event** представляет интерфейс **IlluminateContractsEventsDispat**
- Фасад **DB** - **IlluminateDatabaseConnectionInterface**.

Вообще, последнее не совсем правда. Фасад **DB** представляет **IlluminateDatabaseDatabaseManager**, который содержит вот такую вот магию:

```
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);
    }
}
```

Но нам будет достаточно только использования **ConnectionInterface** напрямую.

```
class SurveyService
{
    public function __construct(
        private ConnectionInterface $connection,
        private Dispatcher $dispatcher
    ) {}

    public function create(SurveyCreateDto $dto)
    {
        if(count($dto->options) < 2) {
            throw new BusinessException(
                "Please provide at least 2 options");
        }

        $survey = new Survey();

        $this->connection->transaction(function() use ($dto, $survey) {
            $survey->title = $dto->title;
```

```
            $survey->save();

            foreach ($dto->options as $option) {
                $survey->options()->create([
                    'text' => $option,
                ]);
            }
        });

        $this->dispatcher->dispatch(new SurveyCreated($survey->id));
    }
}
```

Хорошо. Для интерфейса **ConnectionInterface** я могу создать фейк класс **FakeConnection**. Класс **EventFake**, который используется, когда происходит вызов **Event::fake()**, может быть использован напрямую.

```
use Illuminate\Support\Testing\Fakes\EventFake;
//...

class SurveyServiceTest extends TestCase
{
    public function testCreateSurvey()
    {
        $eventFake = new EventFake(
            $this->createMock(Dispatcher::class));

        $postService = new SurveyService(
            new FakeConnection(), $eventFake);

        $postService->create(new SurveyCreateDto(
            'test title',
            ['option1', 'option2']));

        $eventFake->assertDispatched(SurveyCreated::class);
```

```
    }
}
```

Этот тест выглядит очень похоже на прошлый, с фасадами, но теперь он намного строже с зависимостями класса **SurveyService**. Но не совсем. Любой разработчик всё еще может использовать любой фасад внутри класса **SurveyService** и тест будет продолжать работать. Это происходит потому, что здесь используется специальный базовый класс для тестов, предоставляемый Laravel, который полностью настраивает рабочее окружение.

```
use Illuminate\Foundation\Testing\TestCase as BaseTestCase;

abstract class TestCase extends BaseTestCase
{
    use CreatesApplication;
}
```

Для unit-тестов это недопустимо, нужно использовать обычный базовый класс PHPUnit:

```
abstract class TestCase extends \PHPUnit\Framework\TestCase
{
}
```

Теперь, если кто-то добавит вызов фасада, тест упадёт с ошибкой:

```
Error : Class 'SomeFacade' not found
```

Отлично, от laravel-фасадов мы код полностью избавили.

Отсоединяем от базы данных

Отсоединять от базы данных намного сложнее. Создадим класс репозитория (шаблон **Репозиторий**), чтобы собрать в нём всю работу с базой данных.

```
interface SurveyRepository
{
    //... другие методы

    public function save(Survey $survey);

    public function saveOption(SurveyOption $option);
}

class EloquentSurveyRepository implements SurveyRepository
{
    //... другие методы

    public function save(Survey $survey)
    {
        $survey->save();
    }

    public function saveOption(SurveyOption $option)
    {
        $option->save();
    }
}

class SurveyService
{
    public function __construct(
        private ConnectionInterface $connection,
        private SurveyRepository $repository,
        private Dispatcher $dispatcher
    ) {}

    public function create(SurveyCreateDto $dto)
    {
        if(count($dto->options) < 2) {
            throw new BusinessException(
```

```
                "Please provide at least 2 options");
        }

        $survey = new Survey();

        $this->connection->transaction(function() use ($dto, $survey) {
            $survey->title = $dto->title;
            $this->repository->save($survey);

            foreach ($dto->options as $optionText) {
                $option = new SurveyOption();
                $option->survey_id = $survey->id;
                $option->text = $optionText;

                $this->repository->saveOption($option);
            }
        });

        $this->dispatcher->dispatch(new SurveyCreated($survey->id));
    }
}

class SurveyServiceTest extends \PHPUnit\Framework\TestCase
{
    public function testCreateSurvey()
    {
        $eventFake = new EventFake(
            $this->createMock(Dispatcher::class));

        $repositoryMock = $this->createMock(SurveyRepository::class);

        $repositoryMock->method('save')
            ->with($this->callback(function(Survey $survey) {
                return $survey->title == 'test title';
            }));
```

```
        $repositoryMock->expects($this->at(2))
            ->method('saveOption');

        $postService = new SurveyService(
            new FakeConnection(), $repositoryMock, $eventFake);

        $postService->create(new SurveyCreateDto(
            'test title',
            ['option1', 'option2']));

        $eventFake->assertDispatched(SurveyCreated::class);
    }
}
```

Это корректный unit-тест. Класс **SurveyService** был протестирован в полной изоляции, не касаясь среды Laravel и базы данных. Но почему это не радует меня? Причины в следующем:

- Я был вынужден создать абстракцию с репозиторием исключительно для того, чтобы написать unit-тесты. Код класса **SurveyService** без него выглядит в разы читабельнее, что весьма важно. Это похоже на шаблон **Репозиторий**, но не является им. Он просто пытается заменить операции Eloquent с базой данных. Если объект опроса будет иметь больше отношений, то придётся реализовывать методы **save%ИмяОтношения%** для каждого из них.
- Запрещены почти все операции Eloquent. Да, они будут работать корректно в реальном приложении, но не в unit-тестах. Раз за разом разработчики будут спрашивать себя - “а для чего нам эти unit-тесты?”
- С другой стороны, такие unit-тесты очень сложные. Их сложно писать и сложно читать. Притом, что это пример один из простейших - просто создание одной сущности с отношениями.
- Каждая добавленная зависимость заставит переписывать все unit-тесты. Это делает поддержку таких тестов весьма

трудоемким занятием.

Это сложно измерить, но мне кажется, что польза от таких тестов намного меньше усилий затрачиваемых на них и урону читабельности кода. В начале этой главы я сказал, что Unit-тесты - одни из лучших индикаторов качества кода в проекте. Если код сложно тестировать, скорее всего он обладает высокой связанностью. Класс **SurveyService** точно обладает. Он содержит основную логику (проверку количества вариантов ответа и создание сущности), а также логику приложения (транзакции базы данных, генерация событий, запросы к API и т.д.). Это можно исправить выделив из класса эту самую основную логику. Об этом мы поговорим в следующей главе.

Стратегия тестирования приложения

В этом разделе я не хочу говорить про большие компании, которые создают или уже имеют стратегии тестирования до того, как проект начался. Разговор будет про мелкие проекты, которые начинают расти. В самом начале проект тестируется членами команды, которые просто используют приложение. Время от времени, менеджер или разработчики открывают приложение, выполняют в нём некоторые действия, проверяя корректность его работы и насколько красив его интерфейс. Это неавтоматическое тестирование без какой-либо стратегии.

Дальше (обычно после каких-нибудь болезненных ошибок на продакшене) команда решает что-то изменить.

Первое, очевидное, решение - нанять ручного тестировщика. Он будет тестировать новый функционал, а также может описать главные сценарии работы с приложением и после каждого обновления проходить по этим сценариям, проверяя, что приложение работает как требуется. Ключевые слова - smoke тесты, регрессионное тестирование.

Если приложение продолжит расти, то количество сценариев тоже будет расти. В то же время, команда наверняка начнет чаще выкатывать обновления и вручную проверять каждый сценарий при каждом обновлении станет невозможно. Решение - писать автоматические тесты. Сценарии использования, написанный ручным тестировщиком могут быть сконвертированы в автоматические тесты для Selenium или других инструментов функционального тестирования.

С точки зрения пользователя вашего приложения, функциональное тестирование является самым важным и весьма желательно уделить ему достаточное внимание. Тем более, что если ваше приложение - это API, то писать функциональные тесты к нему - одно удовольствие.

А что же unit-тесты? Да, они могут помочь нам проверить много специфических случаев, которые сложно будет покрыть функциональными тестами, но главная их задача - помогать нам писать код. Помните пример с **cutString** в начале главы? Вопреки распространенному мнению, писать такой код с тестами чаще быстрее, чем без них. Тесты сразу же проверят код на правильность, проверят поведение кода в краевых случаях и в дальнейшем не позволят изменениям в коде нарушить требования к этому коду. Написание unit-тестов должно быть простым. Они не должны быть тяжелым камнем на шее проекта, который постоянно хочется выбросить. В коде наверняка есть много чистых функций, и писать их с помощью тестов - весьма хорошая практика.

Unit-тесты же для контроллеров или Слоя Приложения, как мы уже убедились ранее, писать весьма неприятно, а поддерживать - тем более. “Форсированно интеграционные” тесты проще, но они могут быть не очень стабильны. Если ваш проект имеет основную логику (не связанную с базой данных), которую вы очень хотите покрыть unit-тестами, чтобы, например, проверить поведение при краевых случаях, это весьма толстый намёк на то, что основная логика проекта выросла настолько, что нуждается в отделении от Слоя Приложения. В свой собственный слой.

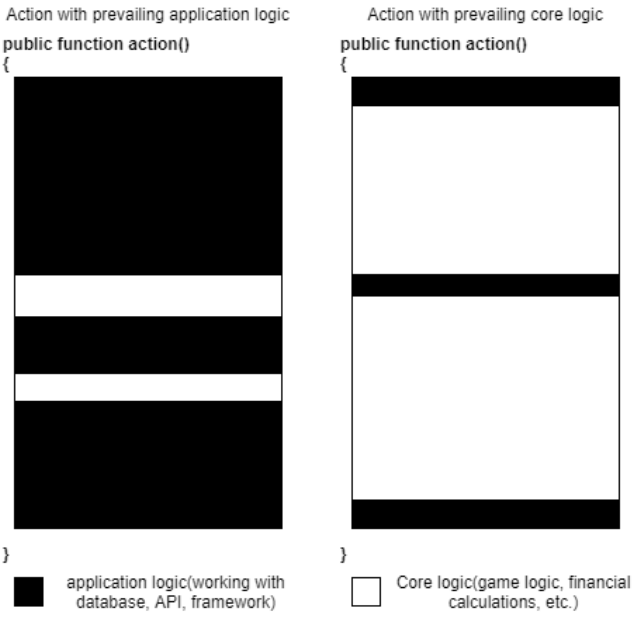
10. Доменный слой

Когда и зачем?

Бизнес-логика, на английском Domain logic или Логика предметной области, это та логика, которую представляют себе пользователи или заказчики, если полностью выкинуть из головы интерфейс пользователя. Например, для игры это будет полный свод её правил, а для финансового приложения - все сущности, которые там есть и все правила расчетов. Для блога всю бизнес-логику можно грубо описать так: есть статьи, у них есть заголовок и текст, администратор может их создать и опубликовать, а другие пользователи могут видеть все опубликованные статьи.

Сложные приложения могут быть разделены на два класса:

- Сложная логика приложения, но средняя или даже простая бизнес-логика. Пример: нагруженные социальные сети или иные контент-проекты. Логика приложения там огромная: SEO, интеграции с API, авторизации, поиск контента и т.д. Бизнес же логика, в сравнении с этим, довольно проста - посты и комментарии.
- По-настоящему сложная бизнес-логика и сложная или не очень сложная логика приложения. Пример: игры или энтерпрайз приложения.



Черным цветом здесь обозначена логика приложения, белым - бизнес-логика. Легко догадаться, что вынесение бизнес-логики в отдельный слой чуть ли не обязательно для приложений второго типа. Для приложений первого типа это тоже может оказаться полезным, но это будет не так просто доказать.

Вынесение бизнес-логики в отдельный слой - весьма серьезная задача, особенно для уже реализованного приложения, которое использует для работы с базой данных Eloquent или другие реализации шаблона Active Record. Связь объектов с базой данных в них жесткая и неразрывная. Их придётся полностью оторвать от базы данных, реализовав их как независимые объекты, реализующие доменную логику. Эта задача может потребовать колоссальных усилий и времени, поэтому решение создавать Доменный слой должно иметь серьезные причины. Я попробую перечислить некоторые из них и они должны быть положены гирьками на воображаемые весы в голове архитектора проекта. На другой стороне весов должна быть большая гиря из времени, пота и крови разработчиков, которым придется многое пе-

реосмыслить, если до этого вся бизнес-логика, реализовывалась объектами, представляющими собой строчки из таблиц базы данных.

Unit-тестирование

В предыдущей главе мы выяснили, что писать unit-тесты для слоя приложения - очень трудно и не очень полезно. Поэтому для приложений с не очень сложной бизнес-логикой, такие тесты почти не пишут. Функциональные тесты для таких приложений намного важнее.

С другой стороны, сложную логику покрывать функциональными тестами - очень дорого по времени разработчиков, а соответственно и по деньгам. Время тратится как на написание таких тестов, так и на выполнение их. Для крупных приложений весь набор функциональных тестов может выполняться часами. Писать же сложную логику с помощью unit-тестов намного эффективнее и продуктивнее, а выполняются они за миллисекунды. Но сами тесты должны быть очень простыми. Давайте сравним unit-тест для объекта, описывающего сущность, а не строчку в базе данных, с unit-тестами для слоя приложения. Последние монструозные тесты из предыдущей главы я копировать не стану, а тесты для сущности Post - легко:

```
class CreatePostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulCreate()
    {
        $post = new Post('title', '');

        $this->assertEquals('title', $post->title);
    }

    public function testEmptyTitle()
    {
```

```
        $this->expectException(InvalidArgumentException::class);

        new Post('', '');
    }
}

class PublishPostTest extends \PHPUnit\Framework\TestCase
{
    public function testSuccessfulPublish()
    {
        $post = new Post('title', 'body');

        $post->publish();

        $this->assertTrue($post->published);
    }

    public function testPublishEmptyBody()
    {
        $post = new Post('title', '');

        $this->expectException(CantPublishException::class);

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

Такие тесты требуют минимальных усилий. Их легко писать и поддерживать. А реализовывать сложную логику сущности при поддержке хорошо написанных тестов - в разы проще. Простота и легкость таких тестов результат того, что класс **Post** реализует единственную ответственность - логика сущности Статья. Он не отвлекается на такие вещи, как база данных. Из подобных классов и состоит Доменный слой.

Простота поддержки кода

Реализация двух логик (бизнес- и приложения-) в одном месте нарушает Принцип Единственной Ответственности. Наказание за это последует довольно быстро. Причем, разумеется, это не будет одним ударом гильотины. Этот искусный палач будет мучать медленно. Каждое движение будет причинять боль. Количество дублированного кода будет расти. Любая попытка вынести какую-либо логику в свой метод или класс встретит большое сопротивление. Две логики, сплетенные в одно целое, всегда будет необходимо отделить друг от друга, перед тем как делать рефакторинг.

Вынесение доменной логики в отдельный слой позволит разработчикам всегда концентрироваться на одной логике, что делает процесс рефакторинга более простым и позволит держать приложение в тонусе, не тратя на это огромное количество времени.

Active record и Data mapper

Eloquent является реализацией шаблона Active Record. Классы моделей Eloquent очень умные - они сами могут загрузить себя из базы и сохранить себя там же. Класс **User**, наследуясь от Eloquent **Model**, наследует огромный пласт кода, который работает с базой данных и сам становится навеки связанным с ней. Работая с ним, всегда приходится держать в голове такие факты как то, что `$this->posts` - это не просто коллекция объектов **Post**. Это псевдосвойство, это проекция отношения `posts`. Нельзя просто взять и добавить туда новый объект. Придется вызвать что-то вроде `$this->posts()->create(...)`.

Этот код не может быть покрыт unit-тестами. Приходится постоянно держать в голове базу данных и когнитивная нагрузка на программиста растёт все больше с ростом сложности логики.

Библиотеки, реализующие шаблон Data mapper, пытаются снять эту нагрузку. Классы сущностей там - обычные классы. Как мы

успели убедиться, unit-тесты для таких классов писать весьма приятно. Когда разработчик хочет сохранить состояние сущности в базе данных, он вызывает метод **persist** у Data Mapper библиотеки и она, используя некоторую мета-информацию про то, как именно нужно хранить эту сущность в базе данных, обеспечивает их преобразование в строчки таблиц и обратно. Самая популярная Data Mapper библиотека в PHP на даный момент - Doctrine. Я буду использовать её для будущих примеров. Следующие “причины” будут описывать преимущества использования чистых сущностей проецируемых в базу данных с помощью Data Mapper, вместо Active Record.

Высокая связность бизнес логики

Вернёмся к примеру с сущностью опроса. Опрос - весьма простая сущность, которая содержит текст вопроса и возможные ответы. Очевидное условие: у каждого опроса должно быть как минимум два варианта ответа. В примере, который был раньше в книге, в действии **создатьОпрос** была такая проверка перед созданием объекта сущности. Это же условие делает сущность **SurveyOption** зависимой. Приложение не может просто взять эту сущность и удалить её. В действии **удалитьВариантОтвета** сначала должно быть проверено, что в объекте **Survey** после этого останется достаточно вариантов ответа. Таким образом, знание о том, что в опросе должно быть как минимум два варианта ответа, теперь содержится в обоих этих действиях: **создатьОпрос** и **удалитьВариантОтвета**. Связность данного кода слабая.

Это происходит из-за того, что сущности **Survey** и **SurveyOption** не являются независимыми. Они представляют собой один **модуль** - опрос с вариантами ответа. Знание о минимальном количестве вариантов ответа должно быть сосредоточено в одном месте - сущности **Survey**.

Я понимаю, что такой простой пример не может доказать важность работы таких сущностей как одно целое. Представим что-

нибудь более сложное - реализацию игры Монополия! Всё в этой игре - это один большой модуль. Игроки, их имущество, их деньги, их положение на доске, положение других объектов на доске. Всё это представляет собой текущее состояние игры. Игрок делает ход и всё, что произойдёт дальше, зависит от текущего состояния. Если он наступает на чужую собственность - он должен заплатить. Если у него достаточно денег - он платит. Если нет - должен получить деньги как-либо, либо сдаться. Если собственность ничья, её можно купить. Если у него недостаточно денег - должен начаться аукцион среди других игроков.

Монополия - отличный пример сложного модуля. Реализация такой огромной логики в методах отдельных сервисных классов приведёт к невероятному дублированию кода и лучший способ это дублирование устранить - сконцентрировать логику в сущностях внутри единого модуля. Это также позволит писать логику вместе с unit-тестами, которые проверят всё огромное множество возможных ситуаций.

Сдвиг фокуса с базы данных к предметной области

Приложение, особенно сложное, не является просто прослойкой между интерфейсом пользователя и базой данных. Однопользовательские игры, которые играются на локальном компьютере (какой-нибудь шутер или RPG), не сохраняют своё состояние в базе данных после каждого происходящего там действия. Игровые объекты просто живут в памяти, взаимодействуя друг с другом, не сохранясь в базу данных каждый раз. Только тогда, когда игрок попросит сохранить игру, она сохраняет всё своё состояние в какой-нибудь файл. Вот так приложение должно работать! То, что web-приложения вынуждены сохранять в базе данных своё состояние после каждого обновляющего запроса - это не удобства ради. Это необходимое зло.

В идеальном мире с приложениями написанными раз и навсегда, работающими на 100% стабильно, на 100% стабильном железе

с бесконечной памятью, лучшим решением было бы хранить всё в объектах в памяти приложения, без сохранения всего в базе данных. Идея выглядит сумасшедшей, особенно для PHP-программистов, которые привыкли, что всё созданное умирает после каждого запроса, но в этом есть определенный смысл. С этой точки зрения, намного выгоднее иметь обычные объекты с логикой, которые даже не думают о таких странных вещах, как таблицы базы данных. Это поможет разработчикам сфокусироваться на главной логике, а не логике инфраструктуры.

Иметь возможность писать логику, используя объекты, и лишь потом решая где и как они будут храниться - весьма полезно. Например, варианты ответов на опрос можно хранить в отдельной таблице, а можно в JSON-поле таблицы с опросами.

Инварианты сущностей

Инвариант класса, это условие на его состояние, которое должно выполняться всегда. Каждый метод, изменяющий состояние сущности, должен сохранять этот инвариант. Примеры: у клиента всегда должен быть email, опрос всегда должен содержать как минимум два варианта ответа. Условие сохранять инвариант в каждом методе, который изменяет состояние, заставляет проверять этот инвариант в этих методах.

Кстати, идея иммутабельных объектов, т.е. объектов, которые создаются раз и не меняют никаких своих значений, делает задачу поддержания инварианта простой. Пример такого объекта: объект-значение **Email**. Его инвариант - содержать всегда только правильное значение строки email-адреса. Проверка инварианта делается в конструкторе. А поскольку он дальше не меняет своего состояния, то в других местах эта проверка и не нужна. А вот если бы он имел метод **setEmail**, то проверку инварианта, т.е. корректности email, пришлось бы вызывать и в этом методе.

В сущностях Eloquent крайне трудно обеспечивать инварианты. Объекты **SurveyOption** - формально независимы, хотя и долж-

ны быть под контролем объекта **Survey**. Любая часть приложения может вызвать **remove()** метод сущности **SurveyOption** и она будет просто удалена. В итоге все инварианты Eloquent сущностей держатся буквально на честном слове: на соглашениях внутри проекта или отлаженном процессе code review. Серьезным проектам необходимо что-то более весомое, чем честное слово. Системы, в которых сам код не позволяет делать больших глупостей, намного более стабильны, чем системы полагающие, что ими будут заниматься только очень умные и высококвалифицированные программисты.

Реализация Доменного слоя

Пример домена

Если вы, тщательно взвесив все За и Против, решили попробовать выделить доменную логику, то давайте начнем с небольшого примера. Фрилансерская биржа - неплохой вариант. Забудем про базы данных и web-интерфейсы. Разрабатываем чистую логику.

```
final class Client
{
    private function __construct(private Email $email) {}

    public static function register(Email $email): Client
    {
        return new Client($email);
    }
}
```

Просто сущность клиента.

Где имя и фамилия клиента?

Я пока не нуждаюсь в этой информации. Кто знает, может это будет биржа с анонимными клиентами? Надо будет - добавим.

Почему тогда есть email?

Мне нужно как-то идентифицировать клиентов.

Где методы setEmail и getEmail?

Они будут добавлены как только я почувствую нужду в них.

```
final class Freelancer
{
    private function __construct(
        private Email $email,
        private Money $hourRate
    ) {}

    public static function register(
        Email $email, Money $hourRate): Freelancer
    {
        return new Freelancer($email, $hourRate);
    }
}
```

Сущность Фрилансер. По сравнению с клиентом было добавлено поле с часовой ставкой. Для денег используется класс **Money**. Какие поля там есть? Что используется для количества: float или integer? Разработчик, который работает над сущностью Фрилансер не должен беспокоиться об этом! **Money** просто представляет деньги в нашей системе. Этот класс умеет всё, что от него требуется для реализации доменной логики: сравнивать себя с другими деньгами и некоторые математические операции.

Изначально проект будет работать с одной валютой и будет хранить денежную информацию в одном поле integer, представляющим собой количество центов (или рублей, неважно). Через несколько лет биржа может стать международной и нужно будет добавить поддержку нескольких валют. В класс **Money** будет

добавлено поле **currency** и изменена логика. В базу данных добавится поле с используемой валютой, в паре мест, где создаются объекты **Money** придётся добавить информацию о валюте, но главная логика, которая использует деньги, не будет даже затронута! Она как использовала объект **Money**, так и будет продолжать.

Это пример принципа **Сокрытия информации**. Класс **Money** предоставляет стабильный интерфейс для концепта денег. Методы **getAmount()**:int и **getCurrency()**:string - плохие кандидаты на стабильный интерфейс. В этом случае клиенты класса будут знать слишком многое о внутренней структуре и каждое изменение в ней будет приводить к большому количеству изменений в проекте. Методы **equalTo**(**Money $other**), **compare**(**Money $other**), **plus**(**Money $other**) и **multiple**(**int $amount**) - прячут всю информацию о внутренней структуре внутри себя. Такие, прячущие информацию, методы являются намного более стабильным интерфейсом. Его не придётся часто менять. Меньше изменений в интерфейсах - меньше хлопот при поддержке.

Дальше, клиент может создать Проект. У проекта есть название, описание и примерный бюджет, но логика работы с ним не зависит от этих данных, важных только для людей. Логика подачи заявок от фрилансеров никак не зависит от заголовка проекта. А вот интерфейс в будущем может поменяться и в проект могут быть добавлены новые поля, которые не влияют на логику. Поэтому, исходя из принципа сокрытия информации, я хочу спрятать информацию о деталях проекта, важных только людям, в объект значение:

```
final class JobDescription
{
    // value object.
    // Job title, description, estimated budget, etc.
}

final class Job
{
    private function __construct(
        private Client $client,
        private JobDescription $description
    ) {}

    public static function post(Client $client,
        JobDescription $description): Job
    {
        return new Job($client, $description);
    }
}
```

Отлично. Базовая структура сущностей создана. Давайте, добавим немного логики. Фрилансер может заявиться делать этот проект. Заявка фрилансера содержит сопроводительное письмо (или просто сообщение заказчику) и его текущую ставку. Он может в будущем поменять свою ставку, но это не должно изменить её в заявках.

```
final class Proposal
{
    public function __construct(
        private Job $job,
        private Freelancer $freelancer,
        private Money $hourRate,
        private string $coverLetter
    ) {}
}

final class Job
{
    //...

    /**
     * @var Proposal[]
     */
    private $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);
    }
}
```

Это другой пример сокрытия информации. Только сущность Фрилансера знает, что у него есть часовая ставка. Каждый объект обладает минимальной информацией, необходимой для его работы. Не должно быть такого, что какой-то объект постоянно дергает поля другого объекта, таким образом зная о нём слишком многое. Система, построенная таким образом - очень стабильна. Новые требования в них часто реализуются изменениями в 1-2 файлах. Если эти изменения не затрагивают интерфейс класса (интерфейсом класса здесь названы все его публичные методы), то изменений в других классах не требуется. Такой код также более защищен от багов, которые могут быть случайно занесены изменениями.

Фрилансер не может добавить новую заявку. Он должен поменять старую для этого. Далее, в базе данных, вероятно, будет добавлен некий уникальный индекс на поля **job_id** и **freelancer_id** в таблице **proposals**, но это требование должно быть реализовано в доменной логике тоже:

```
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(
                    'Этот фрилансер уже оставлял заявку');
            }
        }

        $this->proposals[] = $newProposal;
    }
}
```

Я добавил метод **equals()** в класс **Freelancer**. Как я уже говорил, email нужен для идентификации, поэтому если у двух объектов **Freelancer** одинаковые email - то это один и тот же фрилансер. Класс **Job** начинает знать слишком многое про класс **Proposal**. Весь этот `foreach` - это копание во внутренностях имплементации заявки. Мартин Фаулер назвал эту проблему “завистливый метод” (или Feature Envy в оригинале). Решение простое - перенести эту логику в класс **Proposal**:

```
final class Proposal
{
    //...

    /**
     * @param Proposal $other
     * @throws BusinessException
     */
    public function checkCompatibility(Proposal $other)
    {
        if($this->freelancer->equals($other->freelancer)) {
            throw new BusinessException(
                'Этот фрилансер уже оставлял заявку');
        }
    }
}

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;
    }
}
```

Заявка теперь сама проверяет совместима ли она с новой заявкой. А метод **Proposal::getFreelancer()** больше не используется и может быть удалён.

Инкапсуляцию, которая весьма близка к сокрытию информации, называют одним из трёх китов объектно-ориентированного программирования, но я постоянно вижу неверную её интерпретацию. В стиле “public **$name** - это не инкапсуляция, а private **$name** и методы **getName** и **setName**, банально имитирующие это публичное поле - инкапсуляция, потому, что в будущем можно будет переопределить поведение **setName** и **getName**”. Не знаю как именно можно переопределить эти методы геттера и сеттера, но даже в этом случае всё приложение видит, что у этого класса есть свойство **name** и его можно как прочитать, так и записать, соответственно, будет использовать его везде и интерфейс этого класса не будет стабильным никогда.

Для некоторых классов, таких как DTO, выставлять напоказ своё внутреннее устройство - это нормально, поскольку хранение данных - это единственное предназначение DTO. Но объектно-ориентированное программирование предполагает, что объекты будут работать в основном со своими данными, а не с чужими. Это делает объекты более независимыми, соответственно, менее связанными с другими объектами, и, как следствие, более лёгкими в поддержке.

Я уже успел написать некоторую логику, но совершенно забыл о тестах!

```
class ClientTest extends TestCase
{
    public function testRegister()
    {
        $client = Client::register(Email::create('some@email.com'));

        // Что здесь проверять?
    }
}
```

Сущность клиента не имеет никаких публичных полей и геттеров. Unit-тесты не могут проверить что-либо. PHPUnit не любит когда проверки отсутствуют, он вернёт ошибку: “This test did not perform any assertions”. Я мог бы создать геттер-метод **getEmail()** и проверить, что сущность есть и у неё **email** тот же, который мы передали методу **register**. Но этот геттер-метод, который благодаря принципу сокрытия информации не потребовался в реализации бизнес-логики, будет использоваться только в тестах, что меня совсем не устраивает. Будучи добавленным, он может соблазнить слабого духом разработчика использовать его и в логике, что нарушит гармонию данного класса.

Доменные события

Самое время вспомнить про доменные события. Они в любом случае будут использованы в приложении, просто чуть позже, когда нам, например, понадобится посылать письма после каких-то действий. Они идеальны для тестов, но с ними есть пара проблем.

Когда вся бизнес-логика лежала в слое приложения, сервисный класс спокойно кидал события напрямую в класс Dispatcher, когда ему требуется. Доменный объект так делать не может, поскольку про объект Dispatcher он ничего не знает. Этот Dispatcher объект можно предоставлять доменным объектам, но это может разрушить иллюзию того, что мы моделируем чистую логику.

Как вы заметили, в доменных объектах, которые мы реализовали, речь идёт только о клиентах, фрилансерах и заказах, никаких баз данных, очередей и веб-контроллеров там нет.

Поэтому, более популярным решением является простое агрегирование событий внутри доменных объектов. Простейший вариант:

```
final class Client
{
    //...

    private $events = [];

    public function releaseEvents(): array
    {
        $events = $this->events;
        $this->events = [];

        return $events;
    }

    protected function record($event)
    {
        $this->events[] = $event;
    }
}
```

Сущность записывает события, которые с ней происходят, используя метод **record**. Метод **releaseEvents** возвращает их разом и очищает буфер, чтобы случайно одно событие не было обработано дважды.

Что должно содержать событие **ClientRegistered**? Я ранее говорил, что хочу использовать **email** для идентификации, но в реальной жизни **email** адрес не является хорошим средством идентификации сущностей. Клиенты могут менять их, а также они не

очень эффективны в качестве ключей в базе данных.

Самым популярным решением для идентификации сущностей является целочисленное поле с авто-инкрементным значением, реализованным в движках баз данных. Оно простое, удобное, но выглядит логичным только если доменный слой не отделен полностью от базы данных. Одним из преимуществ чистых доменных объектов является консистентность (или постоянное соответствие инвариантам), т.е. например если у нас есть объект **Client**, то мы можем быть уверены, что у него непустой и корректный **email**, если требования к приложению это предполагают. Любое приложение будет предполагать, что у объекта всегда есть непустой и корректный идентификатор, однако в момент, когда сущность будет создана, но еще не будет сохранена в базе, у неё будет пустой идентификатор, а это не даст, например, сгенерировать правильные события, да и другие проблемы могут проявиться.

Я всё еще не сказал, что мне необходим этот идентификатор для написания unit-тестов, но не люблю этот аргумент про unit-тесты. Те, кто пишет их, и так все понимают. Кто не пишет - для них этот аргумент ничего не значит. Проблемы с unit-тестами - это лишь лакмусовая бумажка, лишь некий показатель проблем с дизайном системы в целом или какой-то её части. Если есть проблемы с тестами, надо искать проблемы в основном коде.

Код с предоставлением **id** для сущности:

```
final class Client
{
    protected function __construct(
        private $id,
        private Email $email,
    ) {}

    public static function register($id, Email $email): Client
    {
        $client = new Client($id, $email);
```

```
        $client->record(new ClientRegistered($client->id));

        return $client;
    }
}
```

Событие **ClientRegistered** записывается в именованном конструкторе, поскольку оно имеет бизнес-имя **registered** и означает, что клиент именно зарегистрировался. Возможно, в будущем будет реализована команда импорта пользователей из другого приложения и событие должно генерироваться другое:

```
final class Client
{
    public static function importFromCsv($id, Email $email): Client
    {
        $client = new Client($id, $email);
        $client->record(new ClientImportedFromCsv($client->id));

        return $client;
    }
}
```

Генерация идентификатора

Наш код теперь требует, чтобы идентификатор был предоставлен сущностям извне, но как генерировать их? Авто-инкрементная колонка в базе данных делала свою работу идеально и будет трудно заменить её. Продолжать использовать авто-инкрементные значения, реализованные через Redis или Memcached - не самая лучшая идея, поскольку это добавляет новую и довольно большую возможную точку отказа в приложении.

Наиболее популярный не-авто-инкрементный алгоритм для идентификаторов это **Универсальный уникальный**

идентификатор (**Universally unique identifier**, UUID). Это 128-битное значение, которое генерируется одним из стандартных алгоритмов, описанных в RFC 4122. Вероятность генерации двух одинаковых значений стремится к нулю. В php есть пакет для работы с UUID - **ramsey/uuid**. Там реализованы некоторые алгоритмы из стандарта RFC 4122. Теперь можно писать тесты:

```
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;
    }
}

class Factory
{
    public static function createUuid(): UuidInterface
    {
        return Uuid::uuid4();
    }

    public static function createEmail(): Email
    {
        static $i = 0;

        return Email::create("test" . $i++ . "@t.com");
    }
}

class ClientTest extends UnitTestCase
```

```
{
    public function testRegister()
    {
        $client = Client::register(
            Factory::createUuid(),
            Factory::createEmail()
        );

        $this->assertEventsHas(ClientRegistered::class,
            $client->releaseEvents());
    }
}
```

Помните тестирование методом черного и белого ящиков? Это пример тестирования методом черного ящика. Тест ничего не знает о внутренностях объекта. Не лезет читать его **email**. Он просто проверяет, что на данную команду в сущности сгенерировалось ожидаемое событие. Другой тест:

```
class Factory
{
    public static function createClient(): Client
    {
        return Client::register(self::createEmail());
    }

    public static function createFreelancer(): Freelancer
    {
        return Freelancer::register(
                    self::createEmail(), ...);
    }
}

class JobApplyTest extends UnitTestCase
{
    public function testApply()
```

```
    {
        $job = $this->createJob();
        $freelancer = Factory::createFreelancer();

        $freelancer->apply($job, 'cover letter');

        $this->assertEventsHas(FreelancerAppliedForJob::class,
            $freelancer->releaseEvents());
    }

    public function testApplySameFreelancer()
    {
        $job = $this->createJob();
        $freelancer = Factory::createFreelancer();

        $freelancer->apply($job, 'cover letter');

        $this->expectException(
            SameFreelancerProposalException::class);

        $freelancer->apply($job, 'another cover letter');
    }

    private function createJob(): Job
    {
        return Job::post(
            Factory::createUuid(),
            Factory::createClient(),
            JobDescription::create('Simple job', 'Do nothing'));
    }
}
```

Эти тесты просто описывают требования к бизнес-логике системы, проверяя стандартные сценарии: заявка фрилансера на проект, повторная заявка фрилансера на проект. Никаких баз данных, моков, стабов... Никаких попыток залезть внутрь, допустим

массива заявок на проект и проверить что-то там. Можно попробовать попросить обычного человека, знающего английский язык, но не разработчика, почитать данные тесты и большинство вещей он поймёт без дополнительной помощи! Такие тесты легко писать и не составит больших усилий писать их вместе с кодом, как того требуют практики TDD.

Я понимаю, что данная логика слишком проста, чтобы показать преимущества отдельного, чисто доменного кода и удобного unit-тестирования, но попробуйте представить себе игру Монополия и её реализацию. Это может помочь ощутить разницу подходов. Сложную бизнес-логику намного проще писать и поддерживать, если рядом верные unit-тесты, а инфраструктурная составляющая приложения (база данных, HTTP, очереди т .д.) отделена от главной логики.

Создание хорошей модели предметной области весьма нетривиальная задача. Я могу порекомендовать две книги: красную и синюю. Выбирать не надо. Можно прочитать обе. **“Domain-Driven Design: Tackling Complexity in the Heart of Software”** от **Eric Evans** и **“Implementing Domain-Driven Design”** от **Vaughn Vernon**. На русском они тоже есть. Книги эти довольно трудны в освоении, но там приводится довольно много примеров из реальной практики, которые могут поменять ваше представление о том, как надо строить модели предметной области. Новые понятия, такие как **Aggregate root**, **Ubiquitous language** и **Bounded context** помогут вам по-другому взглянуть на свой код, хотя первые два из них я совсем немного рассмотрел и в этой книге.

Маппинг модели в базу данных

После построения модели самое время подумать о базе данных, в которой нам надо хранить наши сущности и объекты-значения. Я буду использовать ORM-библиотеку **Doctrine** для этого. Для Laravel есть пакет **laravel-doctrine/orm**, который содержит всё необходимое для использования её в проектах.

Doctrine позволяет использовать разные пути конфигурации маппинга. Пример с обычным массивом:

```
return [
    'App\Article' => [
        'type'   => 'entity',
        'table'  => 'articles',
        'id'     => [
            'id' => [
                'type'     => 'integer',
                'generator' => [
                    'strategy' => 'auto'
                ]
            ],
        ],
        'fields' => [
            'title' => [
                'type' => 'string'
            ]
        ]
    ]
];
```

Некоторые разработчики предпочитают держать код сущностей как можно более чистым и такая внешняя конфигурация отлично подходит. Но большинство разработчиков используют атрибуты - специальные пометки в виде объектов перед классами, полями и другими частями кода. Пример с атрибутами:

```php
use Doctrine\ORM\Mapping AS ORM;

#[ORM\Embeddable]
final class Email
{
    #[ORM\Column(type="string")]
    private string $email;
    //...
}

#[ORM\Embeddable]
final class Money
{
    #[ORM\Column(type="integer")]
    private int $amount;
    // ...
}

#[ORM\Entity]
final class Freelancer
{
    #[ORM\Embedded(Email::class)]
    private Email $email;

    #[ORM\Embedded(Money::class)]
    private Money $hourRate;
}

#[ORM\Entity]
final class Job
{
    #[ORM\ManyToOne(targetEntity="Client")]
    #[ORM\JoinColumn(nullable=false)]
    private Client $client;

    //...
```

```
}
```

Я использую атрибут **Embeddable** для объектов значений и атрибут **Embedded**, чтобы использовать их в других классах. Каждый атрибут это скрытый вызов new и конструктор класса может иметь параметры. Атрибут **Embedded** требует имя класса, и опциональный параметр **columnPrefix**, который полезен для генерации колонок (отлично подходит для случаев адресов **fromCountry**, **fromCity**, ... - значение **from**, и **to** для, соответственно, **toCountry**, **toCity**, ...).

Есть также атрибуты для различных отношений: один ко многим (one to many), многие ко многим (many to many), и т.д.

```
#[ORM\Entity]
final class Freelancer
{
    #[ORM\Id]
    #[ORM\Column(type="uuid", unique=true)]
    #[ORM\GeneratedValue(strategy="NONE")]
    private UuidInterface $id;
}

#[ORM\Entity]
final class Proposal
{
    #[ORM\Id]
    #[ORM\Column(type="integer", unique=true)]
    #[ORM\GeneratedValue(strategy="AUTO")]
    private int $id;
}
```

Каждая сущность имеет поле с идентификатором. Тип `uuid` означает строку в 36 символов, в которой UUID будет храниться в своём стандартном строковом представлении, например `e4eaaaf2-d142-11e1-b3e4-080027620cdd`. Альтернативный вариант

(немного более оптимальный по скорости): 16-байтное бинарное представление.

Сущность **Proposal** (заявка) является частью модуля **Job** (проект) и они никогда не будут создаваться как-либо отдельно. По большому счёту, им не нужен **id**, но **Doctrine** требует его для каждой сущности, поэтому можно добавить авто-инкрементное значение здесь.

Когда **Doctrine** достаёт сущность из базы данных, она создаёт объект требуемого класса без использования конструктора и просто заполняет его значениями из базы данных. Всё это работает через механизм рефлексии PHP. Как результат: объект “не чувствует” базу данных. Его жизненный цикл - естественный.

```
$freelancer = new Freelancer($id, $email);

$freelancer->apply($job, 'some letter');

$freelancer->changeEmail($anotherEmail);

$freelancer->apply($anotherJob, 'another letter');
```

Между каждой этой строкой есть как минимум одно сохранение в базу и большое количество считываний из неё. Всё это может происходить на разных серверах, но объект ничего не чувствует, не вызываются конструкторы или сеттеры, он живёт также как в каком-нибудь unit-тесте. **Doctrine** выполняет огромную инфраструктурную работу для того, чтобы объекты жили без забот.

Миграции

Пора создать нашу базу данных. Можно использовать миграции Laravel, но **Doctrine** предлагает еще немного магии: свои миграции. После установки пакета **laravel-doctrine/migrations** и запуска команды `php artisan doctrine:migrations:diff`, будет создана миграция:

```
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');
}
}
```

Я использовал sqlite для этого тестового проекта. Да, выглядит уродливо по сравнению с красивыми и стройными миграциями Laravel, но **Doctrine** может создавать их автоматически! Команда `doctrine:migrations:diff` анализирует текущую базу данных и метаданные сущностей и генерирует миграцию, которая приведёт базу данных в состояние, когда она будет содержать все нужные таблицы и поля.

Я думаю, достаточно про **Doctrine**. Она действительно позволяет разработчикам строить чистую модель предметной области и эффективно маппить её на базу данных. Изъянов у нее тоже хватает, но для нашего примера они несущественны. Как я уже написал ранее, после вынесения доменной логики, слою приложения остаётся только оркестрация между инфраструктурой и доменом. Теперь его основная задача - связать эти два куска кода в одно целое.

```
final class FreelancersService
{
    /** @var ObjectManager */
    private $objectManager;

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

    public function __construct(
        ObjectManager $objectManager,
        MultiDispatcher $dispatcher)
```

```
    {
        $this->objectManager = $objectManager;
        $this->dispatcher = $dispatcher;
    }

    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();
    }
    //...
}
```

Здесь UUID генерируется в слое приложения. Это может произойти и немного ранее. Я слышал о проектах, которые просят клиентов их API сгенерировать идентификатор для новых сущностей.

```
POST /api/freelancers/register
{
    "uuid": "e4eaaaf2-d142-11e1-b3e4-080027620cdd",
    "email": "some@email.com"
}
```

Этот подход выглядит каноничным. Клиенты просто просят приложение сделать определённое действие и предоставляют все необходимые для этого действия данные. Просто “200 Ok” ответа

достаточно. Значение идентификатора у них уже есть, они могут продолжить работу.

ObjectManager::persist кладёт сущность в очередь на сохранение в базу данных. **ObjectManager::flush** сохраняет все объекты в очереди. Посмотрим на несоздающее действие:

```
final class JobApplyDto
{
    public function __construct(
        public readonly UuidInterface $jobId,
        public readonly UuidInterface $freelancerId,
        public readonly string $coverLetter,
    ) {}
}

final class JobApplyRequest extends FormRequest
{
    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', '')
        );
    }
}
```

JobApplyDto - простое DTO для заявки на проект. **JobApplyRequest** - это объект запроса, который производит необходимую валидацию и создаёт объект **JobApplyDto**.

```
final class FreelancersController extends Controller
{
    public function __construct(
        private FreelancersService $service
    ) {}

    public function apply(JobApplyRequest $request)
    {
        $this->service->apply($request->getDto());

        return /*success*/;
    }
}
```

Контроллеры очень просты. Они лишь передают данные из объектов-запросов в сервисные классы.

```
final class FreelancersService
{
    public function apply(JobApplyDto $dto)
    {
        $freelancer = $this->objectManager
            ->findOrFail(Freelancer::class, $dto->getFreelancerId());

        $job = $this->objectManager
            ->findOrFail(Job::class, $dto->getJobId());

        $freelancer->apply($job, $dto->getCoverLetter());

        $this->dispatcher->multiDispatch(
                    $freelancer->releaseEvents());
```

```
        $this->objectManager->flush();
    }
}
```

Слой приложения тоже довольно простой. Он просит объект **objectManager** достать необходимую сущность из базы данных и производит необходимое бизнес-действие. Главная разница с **Eloquent** - это метод **flush**. Слой приложения не должен просить сохранить каждый изменённый объект в базу. **Doctrine** запоминает все объекты, которые она достала из базы и способна распознать какие изменения произошли с момента загрузки. Метод **flush** анализирует эти изменения и сохраняет все измененные объекты в базу.

В этом конкретном случае **Doctrine** находит, что новый объект был добавлен в свойство **proposals** и создаст новую строку в нужной таблице с нужными данными. Эта магия позволяет нам писать без оглядки на базу данных. Можно просто изменять значения полей, добавлять или удалять объекты из отношений, даже глубоко внутри первично загруженного объекта - всё это будет сохранено в методе **flush**.

Разумеется, всё имеет свою цену. Код **Doctrine** намного более сложный, чем код **Eloquent**. Когда я работаю с последним, при любых проблемах я могу посмотреть его код и найти ответы там. Не могу сказать подобного про **Doctrine**. Её более сложная конфигурация и непростые запросы посложнее обычного “найти по идентификатору” прилично добавляют сложности при поддержке. Не забывайте, что это разные ORM, написанные с разными целями.

Вы можете ознакомиться с полным исходным кодом данного примера здесь:

https://github.com/adelf/freelance-example[1]

Там реализован шаблон CQRS, поэтому лучше сначала прочитать следующую главу.

[1] https://github.com/adelf/freelance-example

Обработка ошибок в доменном слое

В главе “Обработка ошибок” я предложил использовать непроверяемое исключение **BusinessException** для уменьшения количества тегов **@throws**, но для сложной модели предметной области это не самая лучшая идея. Исключения могут быть сгенерированы глубоко внутри модели и на каком-то уровне на них можно реагировать. Даже в нашем простом случае генерируется исключение внутри объекта **Proposal**, при повторной заявке от того же фрилансера, но объект заявки не знает контекста. В случае добавления новой заявки это исключение отправляется наверх обрабатываться в глобальном обработчике. В другом случае вызывающий код может захотеть узнать что конкретно пошло не так:

```
// BusinessException превращается в проверяемое
abstract class BusinessException
    extends \Exception {...}

final class SameFreelancerProposalException
    extends BusinessException
{
    public function __construct()
    {
        parent::__construct(
            'Этот фрилансер уже оставлял заявку');
    }
}

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;
    }
}
```

Другое условие может добавиться с новыми требованиями: делать некий аукцион и не позволять фрилансерам добавлять заявку с часовой ставкой выше, чем текущая (не самое умное требование, но бывает и хуже). Новое исключение будет добавлено в теги **@throws**. Это приведет к каскаду изменений в вызывающем коде.

Другая проблема: иногда вызывающему коду нужно узнать

все возможные проблемы: случаи “тот же самый фрилансер” и “слишком высокая часовая ставка” могут произойти одновременно, но исключение возможно только одно. Создавать какой-то композитный тип исключения, чтобы собирать там все проблемы, сделает код более грязным и какой тогда смысл в исключениях?

Поэтому я часто слышу от разработчиков, что они предпочитают нечто похожее на объект **FunctionResult** из главы про обработку ошибок для своей модели предметной области. С его помощью можно вернуть результат в самом удобном для всех виде.

Пара слов в конце главы

Вынесение доменной логики - это большой шаг в эволюции проекта. Намного более практично делать его в самом начале, но архитектор должен оценить сложность этой логики. Если приложение - простой CRUD с очень небольшой дополнительной логикой, от доменного слоя будет мало толку.

С другой стороны, для сложной предметной области - это определённо хороший выбор. Написание чистых доменных объектов вместе с простыми и красивыми unit-тестами - большое преимущество. Довольно сложно осуществить переход с анемичной модели (когда сущности только хранят данные, а работают с ними другие классы) на модель богатую (когда модель сама работает со своими данными), как в коде, так и в голове разработчика, который ни разу не пробовал так думать. По себе знаю. Это как переход с процедурного программирования на объектно-ориентированное. Это требует времени и много практики, но для некоторых проектов оно того стоит.

Как вы, вероятно, заметили методы геттеры (**getEmail**, **getHourRate**) оказались совсем не нужны для описания модели, поскольку принцип сокрытия информации не предполагает их. Если геттер класса **A** используется в классе **B**, то класс **B**

начинает знать слишком многое о классе **A** и изменения там часто приводят к каскадным изменениям в классе **B**, а также многих других. Не стоит этим увлекаться.

К сожалению, кроме кода, приложения также имеют и интерфейс пользователя, и все эти данные, которые мы так хотим спрятать, надо показывать пользователям. Люди хотят видеть имена и часовые ставки. Создавать методы-геттеры только ради этого не очень хочется, но придется. Или нет? Поговорим об этом в следующей главе.

11. CQRS

Чтение и запись - это разные ответственности?

Как мы выяснили в прошлой главе, геттеры совсем не нужны для реализации корректной доменной модели. Сущности и методы, изменяющие их состояния - вполне достаточный набор. Однако, данные необходимо где-то показывать. Неужели нет иного способа, кроме создания методов-геттеров? Давайте попробуем “изобрести” что-нибудь, а для этого мне нужно вспомнить прошлое.

Хранимые процедуры и представления

Первым проектом в моей профессиональной карьере было огромное приложение с логикой, хранимой в базе данных: в тысячах хранимых процедурах и представлениях. Я писал клиент для всего этого на C++. Представление в базе данных, это сохраненный SQL select-запрос, который выглядит как таблица. Использоваться эта “таблица” должна только для чтения, хотя некоторые движки баз данных позволяют туда писать, но это выглядит не очень логично - писать в результат SQL-запроса.

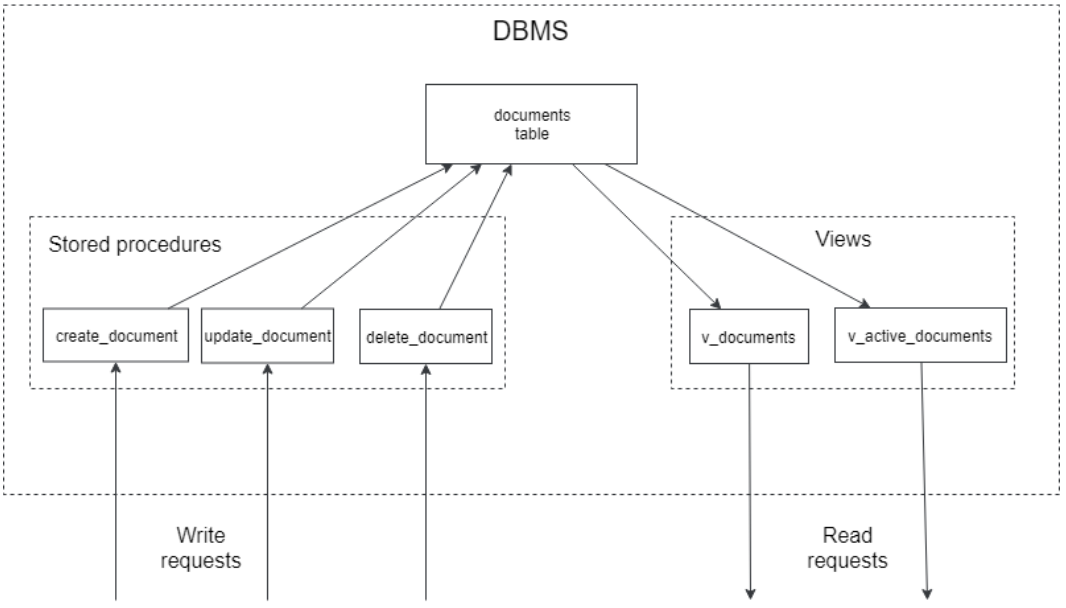
```
CREATE TABLE t (qty INT, price INT);
INSERT INTO t VALUES(3, 50);
CREATE VIEW v AS SELECT qty, price, qty*price AS value FROM t;
SELECT * FROM v;
+------+-------+-------+
| qty  | price | value |
+------+-------+-------+
|    3 |    50 |   150 |
+------+-------+-------+
```

Пример представления из документации MySQL. Оно содержит поле `value`, которое не содержится в таблице.

Хранимая процедура это просто набор инструкций, написанных в процедурном расширении языка SQL (**PL/SQL** в Oracle, **Transact-SQL** в MSSQL, и другие). Это как PHP-функция, которая выполняется внутри базы данных.

```
PROCEDURE raise_salary (
    emp_id NUMBER,
    amount NUMBER
  ) IS
BEGIN
  UPDATE employees
    SET salary = salary + amount
    WHERE employee_id = emp_id;
END;
```

Пример простейшей процедуры в **PL/SQL**. Как я уже говорил, система была огромной, с невероятным количеством логики. Без каких-либо ограничений, такие системы мгновенно превращаются в монструозный неподдерживаемый кусок... кода. Для каждой сущности там была определённая структура процедур и представлений:

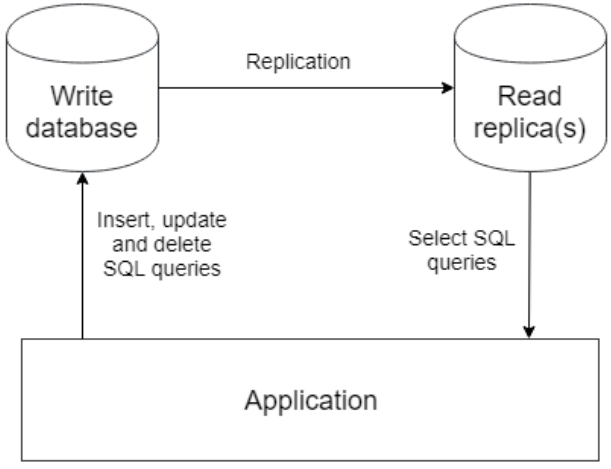


Таблицы там были, как приватные поля класса: трогать их извне было нельзя. Можно было лишь вызывать хранимые процедуры и делать select из представлений. Когда я писал эту книгу, я осознал, что все эти хранимки и представления составляют Слой Приложения. Точно также как слой приложения, описанный мною в предыдущих главах, прячет базу данных от своих клиентов (HTTP, Console и т.д.), эти хранимки и представления прятали реальную таблицу с данными от своих клиентов.

Я вспомнил это всё, потому что здесь очень наглядно показано насколько разными являются операции записи, которые изменяют данные (хранимые процедуры), и операции чтения, нужные для показа данных пользователям (представления). Они используют совершенно разные типы объектов базы данных.

Master-slave репликация

Когда приложение вдруг становится популярным и нагрузка возрастает, первое, что обычно разработчики делают для снижения нагрузки на базу данных, это использование одной базы данных для операций записи и одной (или несколько других) для операций чтения. Это называется master-slave replication.



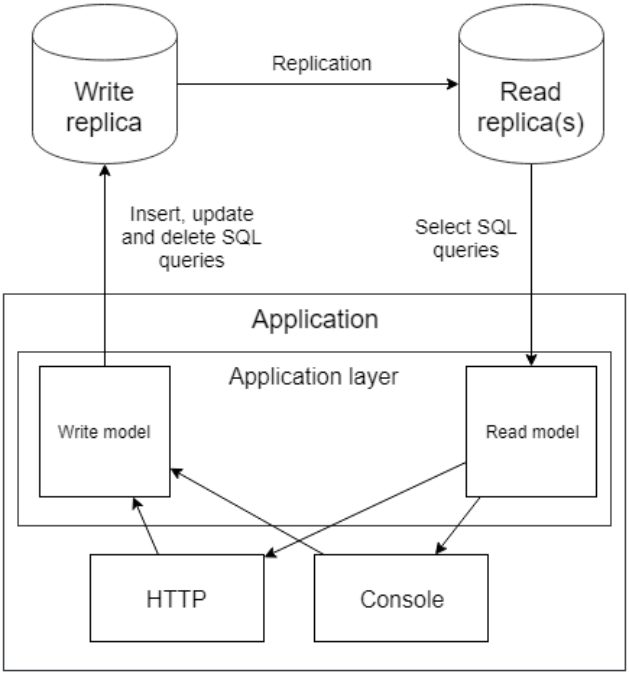
Все изменения идут на главную базу данных и реплицируются на slave базы данных, которые называются репликами. То же самое: write-запросы идут в одно место, read-запросы в другое.

Иногда процесс репликации немного подтормаживает, в силу разных причин, и read-реплики содержат немного старые данные. Пользователи могут изменить какие-то данные в приложении, но продолжать видеть старые данные в нём. Кстати, то же самое происходит тогда, когда в приложении не очень аккуратно реализовано кеширование. Вообще, архитектура системы с одной базой данных и кешем очень похожа на архитектуру приложений с мастер-слейв репликацией. Кеш здесь является подобием read-реплики, которую обновляют вручную из приложения, а не автоматически.

Но любые проблемы с репликацией остаются позади, и реплики

всегда догонят мастер, а кеш протухнет и старые данные заменятся обновлёнными. Т.е. если пользователь изменил какие-либо данные, то он увидит результат, если не сразу, то чуть позже. Этот тип консистентности называется eventual (eventual consistency, по-русски “Согласованность в конечном счёте”). Она - типичный атрибут любых систем с разными хранилищами для записи и чтения.

Разработчики должны всегда держать этот факт в голове. Если выполняется операция записи, все запросы, включая select-запросы, должны идти в хранилище записи. Никаких значений из реплик или кеша. Иначе можно обновить базу данных, используя устаревшие значения. Это условие заставляет разделять слой приложения на две части: код для операций чтения и код для операций записи.



Но это не единственная причина для такого разделения.

Типичный сервисный класс

```
final class PostService
{
    // Методы чтения
    public function getById($id): Post{}
    public function getLatestPosts(): array{}
    public function getAuthorPosts($authorId): array{}

    // Методы записи
    public function create(PostCreateDto $dto){}
    public function publish($postId){}
    public function delete($postId){}
}
```

Обычный сервисный класс для простой сущности. Он состоит из методов для операций чтения и методов для операций записи. Манипуляции и рефакторинги этого класса немного усложнены.

Попробуем реализовать кеширование. Если его реализовать прямо в этом классе, то у него будут как минимум две ответственности: работа с базой данных и кеширование. Самое простое решение - шаблон Декоратор, который я уже применял в главе про внедрение зависимостей. Проблема в том, что для методов записи всё это не нужно: кеширование имеет смысл только для операций чтения. Этот простой факт тоже позволяет осознать, что чтение и запись надо отделять друг от друга.

Попробуем в **PostService** оставить только операции записи:

```
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
{
    public function __construct(
        private PostQueries $baseQueries,
        private Cache $cache,
    ) {}

    public function getById($id): Post
    {
        return $this->cache->remember('post_' . $id,
            function() use($id) {
                return $this->baseQueries->getById($id);
            });
    }
    //...
}
```

Выглядит неплохо! Разделение операций записи и чтения делают рефакторинг и другие манипуляции намного проще, а это говорит о том, что это действие угодно богам.

Отчёты

SQL-запросы для отчётов очень легко показывают разницу в природе запросов чтения и записи. Сложнейшие конструкции из группировок, агрегаций и вычисляемых полей. Когда разработчик пытаются запросить эти данные, используя сущности Eloquent, это выглядит кошмарно. Сущности Eloquent не предназначены содержать агрегированные значения и строить подобные запросы.

Простая идея использовать **язык структурированных запросов** (**Structured Query Language**, **SQL**) быстро приходит в голову. SQL запросы намного более удобны для этой цели. Данные, полученные из этих запросов, можно хранить в простейших классах, как DTO, или просто в массивах. Это еще один пример, когда для одних и тех же данных используются абсолютно разные модели.

Command Query Responsibility Segregation

Шаблон **Command Query Responsibility Segregation**(CQRS) предлагает полностью разделять код на модели чтения и модели записи. Модель здесь - это множество классов, работающих с базой данных: сервисные классы, сущности, объекты-значения и т.д.

Модели для чтения и записи, будучи полностью разделёнными, могут быть реализованы на абсолютно разных технологиях. Write-модель с Доктриной или другим data-mapper и Read-модель с какой-нибудь Active Record библиотекой, а то и просто на чистых SQL-запросах и простейших классах в стиле DTO. Технологии и архитектура для каждой модели выбираются исходя из нужд проекта, без оглядки на другую модель.

Для приложения из прошлой главы с write-моделью, реализованной с помощью Доктрины, read-модель может быть реализована просто с помощью Eloquent:

```
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(
```

```
            "Операция записи недоступна в модели для чтения");
    }
}
```

Базовый класс для Eloquent моделей для чтения. Все операции, которые пишут в базу данных, переопределены с генерацией исключения, чтобы исключить даже теоретическую возможность записать данные с помощью этих классов.

```
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);
    }
}
```

Простейшая реализация. Просто сущности, запрашиваемые напрямую из контроллеров. Как видите, даже со сложной write-моделью некоторые модели для чтения могут быть элементарными, и иногда нет смысла выстраивать какие-либо сложные архитектуры для них. Если нужно, можно реализовать некоторые ***Query**-* или ***Repository**-классы, с кеширующими декораторами и другими шаблонами. Огромным бонусом идёт то, что write-модель не будет даже затронута!

Случаи с простой моделью для записи, но сложной моделью для чтения тоже возможны. Один раз я участвовал в высоконагруженном контент-проекте. Write-модель не была особенно сложной, и она была реализована просто слоем приложения с Eloquent-сущностями. А read-модель содержала много сложных запросов, иногда несколько разных сущностей для одной таблицы, сложное кеширование и т.д. Там были использованы простые SQL-запросы и обычные классы с публичными полями, как read-сущности.

Пара слов в конце главы

Как и любой шаблон, CQRS имеет и преимущества и недостатки. Он позволяет независимо друг от друга разрабатывать модели для записи и чтения. Это позволяет уменьшить сложность модели для записи (удалить геттеры и другую логику, используемую только для чтения) и модели для чтения (использовать простейшие сущности и чистые SQL-запросы). С другой стороны, для большинства приложений это будет дублирование сущностей, некоторой части слоя приложения и т.д. Очевидно, что создание двух моделей одного и того же почти всегда дороже, чем создание одной.

Read- и write-модели часто требуют синхронизации и задача, например “добавить поле в сущность” разбивается на две подзадачи: сделать это для модели чтения и записи. Всё имеет свою цену. И это опять превращается в виртуальные весы в голове архитектора. Здесь я лишь немного описал гирьки на чашах этих весов.

12. Event sourcing

1. e4 - e5
2. Kf3 - Kc6
3. Cb5 - a6

Вы играли в шахматы? Даже если нет, вы знаете эту игру. Два игрока просто двигают фигуры. И это лучший пример для шаблона, про который я хочу поговорить.

Игра королей

Когда любители шахмат хотят узнать про какую-то игру гроссмейстеров, конечная позиция на доске их мало интересует. Они хотят знать про каждый ход!

1. d4 - Kf6
2. Cg5

«Вот это шутка от чемпиона мира!!!»

Смысл текущей позиции на доске полностью зависит от ходов, сделанных ранее:

1. Самое важное: кто ходит следующим? Позиция может одновременно быть выигранной или проигранной в зависимости от этого.
2. Рокировка возможна только если король и ладья не двигались до этого.

3. Взятие на проходе возможно только сразу после того, как пешка сделала ход через клетку.

Давайте создадим приложение для игры в шахматы. Как оно будет хранить игры? Я вижу два варианта:

1. Хранить текущую позицию на доске(т.е. где какие фигуры стоят) с некоторой дополнительной информацией: кто ходит следующим, какие рокировки возможны и некоторая информация о последнем ходе для расчета возможности «взятия на проходе». Все сделанные ходы будут храниться в отдельной таблице, просто для истории.
2. Хранить только сделанные ходы и каждый раз «проигрывать» их для того, чтобы получить текущую позицию.

Как зеркальное отображение этих идей, в шахматном мире существуют две основные нотации:

FEN - хранит текущую позицию на доске со всей необходимой дополнительной информацией. Пример:

```
rnbqkbnr/pp1ppppp/8/2p5/4P3/8/PPPP1PPP/RNBQKBNR w KQkq c6 0 2
```

PGN - просто хранит все ходы. Пример:

```
[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
...(все остальные ходы)...
42. g4 Bd3 43. Re6 1/2-1/2
```

Как видите обе идеи представлены в шахматном мире достаточно широко. Первый путь традиционен для веб-приложений: мы почти всегда просто храним текущее состояние в базе данных. Иногда, имеются также таблицы, где хранится история изменений записей, но эти данные всегда вторичны - это просто некоторые аудит-логи.

Идея хранить только изменения, без текущего состояния выглядит на первый взгляд странной. Каждый раз проматывать все изменения какой-либо сущности, чтобы получить её текущее состояние - это очень медленно, но иметь полную историю игры может быть весьма полезно.

Давайте представим два приложения для игры в шахматы и проанализируем оба пути. Первое приложение будет хранить шахматные игры в такой таблице: id, текущее положение на доске, кто ходит, возможности рокировки, поле для взятия на проходе, если такое есть.

Второе приложение будет просто хранить все ходы с начала партий.

Требования к приложениям постоянно меняются и здесь я нарочно забыл пару шахматных правил про ничью. «Правило 50 ходов» говорит, что должна быть объявлена ничья если произошло 50 ходов с обеих сторон без взятия или хода пешкой. Такое правило добавлено, чтобы избежать бесконечных партий и в компьютерных шахматах ничья объявляется автоматически. Как наши два приложения будут реализовывать данное правило?

Первое приложение должно будет добавить новое поле в таблицу: сколько ходов сделано без взятий и движений пешкой и оно будет вычисляться после каждого хода. Остаётся только проблема с теми партиями, которые идут прямо сейчас, там это поле ещё не посчитано и придётся оставить их без этого правила. Проблема в том, что если вместо шахмат мы возьмём какие-нибудь важные сущности из финансовых или страховых областей, то никто нам не позволит просто оставить старые сущности без новых «правил», потому что эти правила могут быть законами

или важными финансовыми показателями.

Второе приложение просто добавит эту логику в свои алгоритмы и это правило мгновенно станет работать для всех партий, в том числе и для текущих.

Кажется, вторая система готова к изменениям гораздо лучше, но если вы ещё сомневаетесь, то вот вам ещё одно правило про ничьи: тройное повторение. Если позиция на доске повторилась три раза за время партии, то объявляется ничья. Я не знаю каким образом разработчики первого приложения будут реализовывать это требование: им придётся где-то хранить все предыдущие положения на доске. Я часто наблюдаю такую картину: выбран неверный вариант хранения данных, алгоритм или другое архитектурное решение и каждое изменение требований вызывает боль. Код сильно «сопротивляется» изменениям. В ход идут костыли и заплатки. Когда такое происходит, стоит немного отвлечься и попробовать взглянуть на систему по-другому, подвергая сомнению каждое архитектурное решение, сделанное ранее. Вероятно, ещё не поздно его поменять.

Второе приложение опять просто добавит в свою логику это правило и ничего менять в системе хранения партий не придётся. Как видите, в некоторых предметных областях знание о том, что было ранее, очень важно для бизнес-логики и идея хранения всей истории сущности как активного участника логики, а не как пассивных логов, может быть весьма здравой.

Однако такое приложение все ещё трудно представить: история важна для логики, но пользователям обычно интересно только текущее состояние сущностей. Рассчитывать финальное состояние для каждого запроса на чтение в популярном приложении может сильно ударить по производительности. Для решения этой проблемы можно использовать идеи из прошлой главы. Там мы говорили о полностью отделенном коде для работы с операциями записи и чтения. Здесь разными будут и хранилища данных.

Для операций записи будут использоваться хранилище с табли-

цами, которые хранят всю историю сущностей (есть и специальные хранилища оптимизированные для хранения событий - можно погуглить Event store). Для операций чтения будет использоваться традиционное хранилище с таблицами, хранящими текущее состояние. После каждого сделанного хода можно рассчитывать текущее состояние и записывать его в таблицу, которая будет использоваться для операций чтения.

Я выбрал шахматы, поскольку это лучший пример настоящей Event sourcing предметной области. Шаблон Event sourcing предлагает хранить все изменения в системе как последовательность событий. Т.е. вместо традиционной таблицы **posts**:

```
PostId, Title, Text, Published, CreatedBy
```

Всё будет храниться в таблице **post_events**, в которую можно только вставлять новые записи или считывать их, т.е. никаких update или delete запросов - историю нельзя изменять. Выглядеть она будет так:

```
PostId, EventName, EventDate, EventData
```

Вероятные события:

- **PostCreated**(Title, Text, CreatedBy)
- **PostPublished**(PublishedBy)
- **PostDeleted**

Разумеется, блог это явно не та предметная область, для которой стоит реализовывать шаблон Event Sourcing. Очень сложно придумать такую логику с постами в блог, которая зависела бы от их истории. Хранение данных как последовательность событий имеет такие преимущества:

- разработчики могут «продебажить» любую сущность и понять как именно она пришла к своему текущему состоянию.

- состояние всего приложения можно рассчитать на любой момент времени и увидеть как всё было тогда.
- любое состояние, базирующееся на исторических событиях может быть просчитано для любой сущности, в том числе и для старых. Например, если были забыты поля ´created_at´ и ´updated_at´ - их всегда можно рассчитать для всех сущностей позже.
- любая логика, которая зависит от истории сущности, может быть реализована и она заработает немедленно для всех сущностей. Даже созданных до того, как эта логика зародилась в чьей то голове. Пример, некий трекер задач: требование о том, что если задание было назначено на одного и того же пользователя 3 раза - подписать менеджера на эту задачу.

Существует довольно много индустрий, в которых текущее состояние сущностей не является единственными важными данными:

- Баланс вашего банковского счёта всегда рассчитывается как результат всех транзакций с этим счётом, которые банк хранит как основной источник данных.
- Расчеты страховых компаний полностью основаны на истории.
- Медицинские данные всегда лишь некие записи из истории.
- Бухгалтерские программы работают исключительно с произошедшими событиями.

Реальными примерами технологий, использующих подход Event sourcing, являются современные системы хранения кода (git) и блокчейн.

Git хранит данные как последовательности изменений. Изменение, которое чаще называется коммитом, содержит события, такие как: файл А создан с таким вот содержимым, такие-то строки вставлены в содержимое файла Б в такую-то позицию, файл В удалён.

Блокчейном называется последовательность информационных блоков, в которую можно только добавлять и каждый блок содержит криптографический хеш, вычисляемый из предыдущего блока.

Базы данных хранят все операции, которые изменяют данные (insert, update и delete запросы), в специальном логе транзакций и в некоторых ситуациях он используется как главный источник данных. Процесс репликации обычно основан на передаче этого лога транзакций с мастер базы данных в реплики.

Unit-тестирование сущностей

Давайте взглянем снова на unit-тесты для модели, написанные в главе про Доменный слой.

```
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');
    }
}
```

Вместо того, чтобы создавать сущность в нужном нам состоянии и тестировать её поведение в этом состоянии, тесты вынуждены создать сущность в ее изначальном состоянии и, выполняя

некоторые команды, довести её до нужной кондиции. Этот тест повторяет идеи Event sourcing.

События - это единственная информация, которую эти модели отдают наружу и unit-тесты могут проверять только их. Что, если некая сущность вернёт событие об успешном действии, но забудет обновить своё состояние? Тест на данное действие будет успешным. Если сущность хорошо покрыта тестами, то, скорее всего, какие-нибудь другие тесты упадут. Если сущность **Job** не добавит заявку, то тест **testApplySameFreelancer** “упадёт”. Однако для сложных сущностей (помните Монополию и шахматы?), таких тестов может не найтись и сущность с некорректной логикой пройдёт все unit-тесты. Простой пример с публикацией статей:

```
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();
```

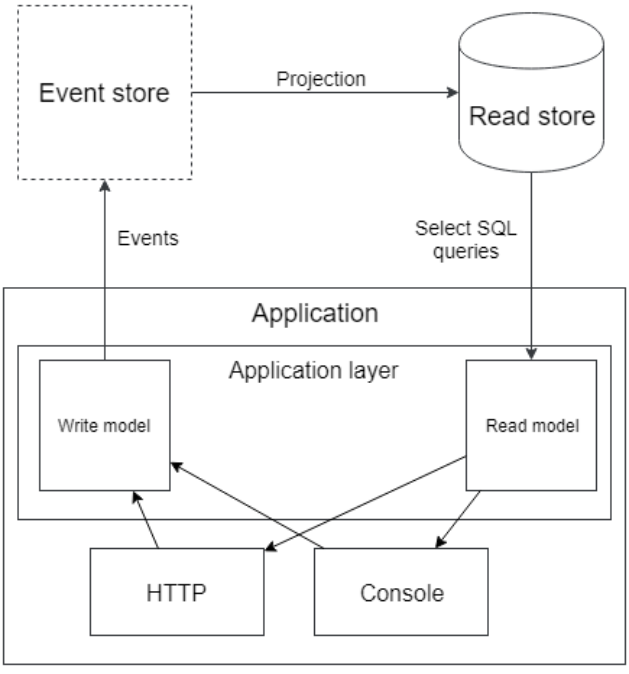
```
        // check
        $this->assertEventsHas(
            PostPublished::class, $post->releaseEvents());
    }
}
```

Тесты будут корректными, но поле в базе не обновится. Пост опубликован не будет и функциональные тесты, если они написаны, должны “упасть”, но надеяться на другие тесты - не самая лучшая стратегия. В Event sourcing (ES) системах события являются главным источником данных, поэтому само событие PostPublished является фактически аналогом записи `$this->published = true;` и unit-тестирование ES-сущностей выглядит намного более естественным. Тесты проверяют реальное поведение.

Мир без магии

Как сущности из главы про Доменный слой сохраняются в базе данных? Классы сущностей там скрывают почти всю информацию о себе, с приватными полями и публичные методы там только для действий, изменяющих состояние. Доктрина анализирует файлы с сущностями, получая мета-информацию о том, как поля должны быть сохранены и в каких таблицах. После команд **persist** и **flush** она использует всю мощь тёмной магии PHP reflection, получая нужные значения и сохраняя их в базе. Что, если магия покинет наш грешный мир? В таком мире будут невозможны Doctrine-сущности.

Можно попробовать забыть про **Сокрытие информации** и сделать всю внутренность сущностей публичной, либо просто реализовать шаблон Event Sourcing! События публичны и полностью открыты. Их можно легко сохранять в базу. Как следствие, системы, обеспечивающие сохранение ES-сущностей в хранилищах, на несколько порядков проще, чем Doctrine и ей подобные.



Архитектура ES-систем повторяет схему CQRS-приложений, но различия моделей для чтения и записи коснулись и хранилищ. Модель для записи, вместо данных, которые могли бы использоваться в модели для чтения, просто хранит события, а данные для чтения - лишь проекция этих событий:

- Традиционное текущее состояние сущностей в таблицах
- Полнотекстовые индексы для поиска (SphinxSearch или Elasticsearch)
- Специальные статистические таблицы для отчётов (они частенько хранятся в отдельных таблицах или базах данных и в традиционных системах)
- Могут быть и другие данные для чтения

События - это достоверный источник всех данных во всех приложениях. Хранение их как первостепенный источник данных - весьма неплохая мысль, однако всё имеет свою цену.

Реализация ES

На момент написания книги, лучшей PHP библиотекой для реализации ES-подхода была **prooph/event-sourcing**, однако недавно я обнаружил обращение одного из разработчиков (полный вариант - https://www.sasaprolic.com/2018/08/the-future-of-prooph-components.html), в котором он объясняет почему они решили отказаться от дальнейшей разработки этой библиотеки. Если помните, я уже говорил, что в данном подходе практически не нужна сложная магия с рефлексией, и разработчик прямо пишет, что для реализации ES-подхода неправильно использовать какую-либо библиотеку, потому что написать те несколько строк кода для реализации подхода будет полезно как с точки зрения понимания ES разработчиками проекта, так и с точки зрения уменьшения зависимостей проекта от других библиотек. Однако это не помешает мне рассмотреть пример использование данной библиотеки, написанный самими разработчиками: **prooph/proophessor-do**.

Каждая сущность имеет Value object для своего id. Это имеет смысл как для сокрытия реального типа, используемого для ключа, так и для того, чтобы случайно в коде не попытаться получить сущность, используя id для другой сущности. Этот подход популярен и в Doctrine-подобных сущностях.

```
interface ValueObject
{
    public function sameValueAs(ValueObject $object): bool;
}

final class TodoId implements ValueObject
{
    private function __construct(private UuidInterface $uuid) {}
```

```
    public static function generate(): TodoId
    {
        return new self(Uuid::uuid4());
    }

    public static function fromString(string $todoId): TodoId
    {
        return new self(Uuid::fromString($todoId));
    }

    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);
    }
}
```

Здесь **TodoId** просто представляет собой UUID значение.

```
final class TodoWasPosted extends AggregateChanged
{
    /** @var UserId */
    private $assigneeId;

    /** @var TodoId */
    private $todoId;

    /** @var TodoText */
    private $text;

    /** @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->text = $text;
        $event->assigneeId = $assigneeId;
        $event->todoStatus = $todoStatus;

        return $event;
    }

    public function todoId(): TodoId {...}

    public function assigneeId(): UserId {...}

    public function text(): TodoText {...}

    public function todoStatus(): TodoStatus {...}
}
```

Событие, возникающее при создании объекта Todo. **AggregateChanged** - это базовый класс для всех ES-событий в **Prooph**. Именованный конструктор используется для того, чтобы код выглядел как естественное предложение на английском: **TodoWasPosted::byUser(...)**. Всё, даже текст и статус, обёрнуты в Value Object классы. Пользу от такого сокрытия информации я приводил в главе про Доменный слой.

Каждая сущность должна наследоваться от класса **AggregateRoot**. Главные его части:

```php
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);
}
```

Тот же самый шаблон хранения событий в сущности, использованный нами ранее. Различия лишь в методе **apply**. ES-сущности могут изменять своё состояние только применяя события. Каждый раз сущность восстанавливает своё состояние, “проигрывая” все события, произошедшие с ней с самого начала.

```
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;
        }
    }
}
```

Небольшая часть сущности **Todo**. Главное различие - состояние сущности полностью зависит от событий. Метод **markAsDone** не меняет состояние напрямую. Только через событие **TodoWasMarkedAsDone**.

Для сохранения сущности используется id и все события, которые с ним произошли с последнего сохранения, получаемые с помощью метода **popRecordedEvents**. Они сохраняются в хранилище событий (это может быть просто таблица в базе данных, или что-то другое). Для выстраивания сущности по id из хранилища получают все события, создают новый объект нужного класса и “проигрывают” все события через него.

```
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();
        }
    }
}
```

Другая часть сущности **Todo**: добавление дедлайна. Просто почитайте код. Он выглядит как простой английский текст, благодаря использованию правильно именованных конструкторов и объектов-значений. Дедлайн - это не просто **DateTime**, а специальный объект **TodoDeadline** со всеми нужными вспомогательными методами, такими как **isInThePast**. Всё это делает клиентский код очень чистым, легким для чтения, что очень важно для больших проектов, разрабатываемых командой программистов.

Не хочу углубляться дальше в этот пример, я рекомендую ознакомиться самим, если интересно - **https://github.com/prooph/proophessor-do**

Проекции - это объекты, которые трансформируют ES-события в данные, удобные для чтения. Практически каждая ES-система имеет проекции, которые выстраивают традиционные таблицы, содержащие текущее состояние сущностей.

```
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
        );
    }
}
```

Этот класс представляет таблицу для содержания текущего состояния todo-задач. Методы **init**, **reset** и **delete** используются когда система хочет создать или пересоздать проекцию. Методы **insert** и **update** очевидно для добавления/изменения записей в таблице. Такой же класс может быть создан для построения полнотекстовых индексов, статистических данных или просто для логирования всех событий в файле (это не самый лучший вариант использования проекций - все события и так хранятся в хранилище событий).

```
$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();
```

Это конфигурация проекции. Она использует класс **TodoReadModel** и трансформирует события в команды для

этого класса. Событие **TodoWasPosted** приводит к созданию новой записи в таблице. Событие **TodoWasMarkedAsDone** изменяет поле статуса для определённого **id**. После трансформа всех событий таблица **read_todo** будет содержать текущее состояние всех todo-задач. Типичный процесс работы с данными в ES-системе такой: получение сущности из хранилища событий, вызов команды (**markAsDone** или **addDeadline**), получение всех новых событий из сущности (их может быть больше одного), сохранение их в хранилище событий, вызов всех проекций. Некоторые проекции хочется вызвать сразу, особенно те, которые изменяют таблицы с текущим состояние. Некоторые можно выполнить отложено в очереди.

Уникальные данные в ES-системах

Одним из недостатков подхода ES является то, что данные в них невозможно проверять условиями, такими как уникальные индексы. В традиционных базах данных, обычный уникальный индекс на колонку `users.email` позволяет нам быть спокойными - в системе не будет двух юзеров с одинаковыми email. Сущности же в ES-системах абсолютно независимы друг от друга. Разные пользователи с одинаковыми email вполне могут жить рядом друг с другом, но требования к системе такого не допускают.

Некоторые приложения используют уникальные индексы из таблиц для чтения, но на 100% такие проверки не защищают. Некоторые системы просто создают специальную таблицу с одним полем для email под уникальным индексом и вставляют значение туда перед попыткой сохранить событие о создании пользователя.

Пара слов в конце главы

Шаблон Event Sourcing - очень мощная вещь. Он позволяет легко реализовывать логику, основанную на исторических данных. Он может помочь приложениям для регулируемых областей иметь правильные аудит-логи. Он также помогает приложениям быть лучше готовыми к изменениям, особенно если эти изменения основаны на том, что происходило с сущностями в прошлом.

Недостатков тоже хватает. ES очень “дорогой” подход. Дорогой по времени кодирования, по железу и времени системных администраторов для обслуживания проекта, по требуемому уровню разработчиков. В команду будет сложнее вливаться новым членам. “Мышление событиями” сильно отличается от “мышления строчками в базе данных”. Анализируя проекты, подобные **proophessor-do**, или создавая свои, особенно для нестандартных предметных областей, можно достичь более глубокого понимания преимуществ и недостатков этого подхода для каждого приложения и более обоснованно выбирать или не выбирать его для новых проектов.

13. Заключение

Эта книга просто некий обзор практик, которые мне показались полезными при разработке приложений. Возможно, кому-то она поможет выбрать нужную для своего проекта. Главное, нужно понять, что она не о том, что надо каждое приложение взять и переписать с использованием Event Sourcing. К каждому приложению нужен свой подход и одни и те же практики отлично подходят к одним приложениям, но будут вредны для других.

Небольшой список литературы, если кому-то захочется углубиться:

Классика

- **“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 и CQRS

- https://aka.ms/cqrs - CQRS journey by Microsoft

- https://github.com/prooph/proophessor-do - Prooph library example project

Unit тестирование

- F.I.R.S.T Principles of Unit Testing