Backend

Оглавление

Общие сведения

Описанное ниже - результат долгого труда. Текущие подходы могут не быть каноническими в индустрии, но помогают нам решать задачи и писать ПО.

В процесс разработки всегда можно (и нужно) приносить что-то новое, однако:

- это должно удовлетворять насущные потребности;
- это должно происходить осмысленно.

Продукт должен развиваться, причём эволюционно.

Рефакторинг необходимо проводить постоянно, чтобы не было мёртвого, дублирующего и рискованного кода. Как только прекращается рефакторинг старого кода, а только пишется новый - качество и бизнес-ценность ПО как продукта только снижается, поскольку его поддержка будет дорожать по мере увеличения кодовой базы.

Поэтому мы пишем код так, чтобы он был понятен коллегам, чтобы его можно было легко поддерживать, развивать и оптимизировать.

Сокращения и термины в рамках статьи

1. **ПО** - программное обеспечение
2. **БП** - бизнес-процесс (не базовая поставка, как в МИС/КДЛ)
3. **БД** - база данных
4. **ПП** - подпрограмма, хранимая процедура или функция, написанная в БД (включая пакеты)
5. **ЖЦ** - жизненный цикл обработки запроса (о работе всего ПО и php-скриптов, составляющих тот или иной backend-сервис)
6. **DTO** - data transfer object, объект переноса данных
7. **VO** - value object, объект-значение
8. **GC** - garbage collector, сборщик мусора
9. **ET** - механизм единой транзакции (о моделях)
10. **Бизнес-слой** - набор классов, которые выполняют бизнес-логику
11. **Расширения, Extensions** - набор классов и кода, который написан внутри команды, не имеет прямого отношения к бизнес-процессам и переиспользуется в разных сервисах для унификации идентичных операций.

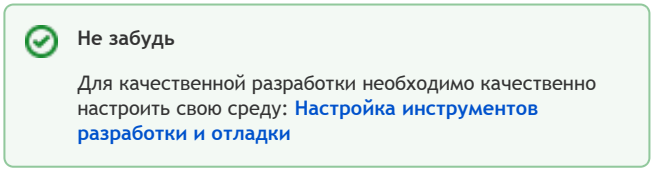
Полезные ссылки и документация

1. PSR - общие стандарты php-разработки к соблюдению
2. Laravel - основной фреймворк (лучшие практики, подсказки)
3. psalm - статический анализатор
4. php_codesniffer - (phpcs) синтаксический анализатор
5. phpunit - фреймворк для тестирования (для Laravel)
6. Каталог паттернов проектирования (примеры на php)
7. OpenRPC: спецификация, песочница
8. JSON-RPC: спецификация
9. JSON Schema: спецификация
10. OpenAPI: спецификация
11. Адель Файзрахманов - «Архитектура сложных веб-приложений. С примерами на Laravel» https://github.com/adelf/acwa_book_ru
12. JWT.io - документация и онлайн-отладка jwt; используемые библиотеки:
    - miladrahimi/php-jwt
    - tymon/jwt-auth
13. DTO, VO:
    - spatie/laravel-data
    - spatie/data-transfer-object (deprecated)
14. Keycloak - сервис авторизации (документация по реализации)

Обрати внимание

Работа со справочниками раскрыта в этих разделах:

- Система кеширования данных (НСИ)
- Уголок Back-End разработчика lis-nsi
- Прогрев справочников (НСИ)



Некоторые принятые архитектурные решения и подходы

Ниже описан наш подход к организации кода. В целом, он устоялся, но может иметь различия, в зависимости от проекта.

Классы

В основном, не финализируем – через это имеем возможность mock-ать их в тестах и расширяться в регионах. Покрываем тестами по максимуму.

Трейты

Избегаем в бизнес-слое. Не возбраняется в расширениях.

Легаси (v1)

Правим по необходимости: исправляем баги и дорабатываем под текущие нужды в рамках задач. Рефакторим осторожно. В целом не развиваем и нового не пишем, но опять же – всё в рамках необходимого и достаточного.

Кеширование

К ресурсам относимся бережно. Если что-то можно не писать в кеш – не пишем. Если кеш больше не нужен – чистим явно (по возможности) и на TTL особо не полагаемся.

Кеш не должен содержать промежуточных этапов БП. Кеш может содержать только данные для переиспользования в разных БП.

Данные, которые не критичны для доступности и использованы в разное время, храним в Redis (lis-redis). Можно тегировать через фасад `Cache`.

Данные, до коих нужно доступаться ультрабыстро, храним в **apcu** (хранится в ОЗУ, долгосрочен, умирает вместе с докер-контейнером) или **array**-кеше (хранится в ОЗУ, краткосрочен, умирает в конце ЖЦ). Тегировать уже не получится.

Касаемо НСИ (lis-nsi) – не так всё просто.

Статические методы, замыкания и функции

Стараемся не избегать, помогаем gc. Если в замыкании/функции /методе нет обращения к **`$this`**, то замыкание обязано быть статическим. Если в замыкании один **`return`**, то замыкание обязано быть стрелочным. Одно не исключает другого.

Однако помни: **статические методы не мокаются в тестах!**

Нейминг

Любые имена должны при минимальной длине нести максимум смысла, чтобы не приходилось объяснять их комментариями. Используем терминологию на английском языке. **Не используем** транслит с русского, венгерскую нотацию и её подобия. Всё согласно PSR-12 и код-стайла (ссылка).

Методы OpenRPC (lis-arm) именуются по простому принципу:
<контроллер><точка><метод>
Пример:

- метод API: api/arm/v2/**Locus.create**
- контроллер: App\Modules\v2\Controllers**Locus**Controller
- модель: App\Modules\v2\Models**Locus**Model
- обработчик запроса: App\Modules\v2\Controllers**Locus**Controller::**create**()

Исключения

Магические методы

Магические значения (скаляры)

Должны явно выбрасываться во всех потенциально рискованных операциях. Прежде всего, это касается операций с БД и БП с некорректными входными данными.

Стараемся избегать, особенно в БП-ах. Явное лучше неявного. По крайней мере, в том коде, который мы пишем и можем контроллировать.

Если назревает необходимость захардкодить какое-то фиксированное скалярное значение, происхождение и назначение коего придётся потом объяснять коллеге, значит нужно **вынести это значение в константу** текущего или вышестоящего класса /интерфейса, **дать ей внятное имя** и объяснить всё в **phpdoc-блоке** к ней.

Если предполагается, что есть некий флаг (признак), который предполагает несколько значений, влияющих на алгоритмы, выборки, расчёты и т.д., значит пишем **enum-класс** (php >= 8.1) и переиспользуем.

Непонятных безымянных значений быть не должно.

Типизация

Строгая. Каждый php-файл начинаем с declare(strict_types=1). Типизируем всё. Используем type hints и return types по максимуму. Пишем дженерики (см. phpdoc, DTO, VO, ещё и ещё). Соответственно, сравнение всегда строгое `===` .

Правда, иногда в контроллерах можно нарваться и на такое.

Переводы сообщений (i18n)

Не путать с локализацией (l10n) – она на текущий момент нам недоступна (?).

Продукт должен поддерживать минимум два языка: русский и английский. Поэтому не следует хардкодить текстовые значения (сообщения), особенно касающиеся БП. Для этого есть директория **resources/lang**. В ней собраны языковые файлы. Все сообщения следует хранить там.

Интерфейсы (контракты, инверсия зависимостей)

По-хорошему, реализация должна опираться именно на контракты, поскольку, например, lis-arm имеет версионность API и "региональность" (где БП и данные могут иметь региональную специфику). Фактически это происходит не всегда (пока не везде есть в этом необходимость).

Тесты

Следует использовать **coverage**-режим xdebug (xdebug.mode) для построения карт покрытия. Само по себе gjrhsnbпокрытие не являтся чем-то обязательным и метрики по нему не отслеживаются, но **это серьёзно помогает** наглядно просмотреть участки кода, которые фактически оказались непротестированными.

Класс теста следует наследовать от **`\Tests\TestCase`** или его наследника, которые, в свою очередь, могут содержать провайдеры данных, дополнительные ассерт-методы и пр.

Хотя технически в рамках phpunit это не имеет значения, но вместо **`self::assert...`** у нас принято писать **`$this->assert...`**
Поскольку мы пишем и используем кастомные ассерты, так код тестов выглядит однообразно.

Для удобства можно настроить запуск тестов внутри контейнера из PHPStorm.

Переиспользуемый код и внешние зависимости

На данный момент, такие модули находятся в директориях **app/Extentions** проектов. Они специально пишутся низкосвязанными, чтобы можно было переносить между проектами без дополнительных правок.

Подход к сторонним зависимостям composer для production следующий: если можно обойтись без них, значит нужно обойтись без них. Подключать мелочёвку нет смысла, на подключение иных пакетов должна быть причина. Как показывает практика, сторонние зависимости у нас подключаются очень редко.

При любом раскладе, при сборе релизной версии весь код проекта пакуется **вместе с зависимостями** в один docker-образ.

Ранний возврат

Методы и функции должны быть оптимальными. Помимо прочего, это значит, что они должны прервать выполнение (и вернуть результат) как можно раньше.

Если входные некорректны, нет смысла что-то вообще делать. Если произошёл сбой в середине – выброси исключение или верни null. Главное – вовремя остановиться.

Внутри **`return`** лучше писать тернарник, если это возможно, чем развесистый **`if`**. Лучше **`return match`**, чем **`switch-case-return`**.

phpdoc

Комментарии внутри кода

Везде и всегда. Писать новые, актуализировать старые. К классам, к методам, к функциям, к переменным, членам класса, константам (!), к чёрту лысому... Ко всему. Без компромиссов.

Если на вход или выход предполагается коллекция или массив каких-то однородных данных - пишем дженерик. Да, это ненастоящие дженерики, они работают только на уровне стат. анализа (psalm, IDE) - это лучше, чем ничего.

Примеры дженериков

```
/**
 * @var array<int>|null  ID
 *   null,
 */
public ?array $research_ids;

/**
 * @var int[]|null  ID
 *
 */
public ?array $research_ids;

/**
 * @var array<string, mixed>
 *   ,   - ,  -
 */
protected array $value_map;

/**
 * ...
 * @param Collection<Indicator> $calculated
 * ...
 */
public function save(Collection $calculated):
array
```

Array vs Illuminate\Support\Collection vs LazyCollection

Коллекции иммутабельны и удобны в использовании. Благодаря чейнингу, можно пересобрать все данные, обогатить, поменять форматы и вложенность данных буквально сразу в **`return`** метода.

Если коллекцию не предполагается кешировать, то лучше всего подойдёт **LazyCollection**. Это может дать существенный прирост в производительности. Там под капотом генераторы, поэтому закешировать её не получится (при сериализации объекта коллекции каждый её элемент коллекции должен быть заведомо известен).

Но применять их следует разумно: если операция банальна (например, вернуть несколько простых объектов или список скаляров), лучше взять **массив**. Плюс, следует помнить про `each` `/map/transform/sort` и прочие методы, которые перебирают всю коллекцию.

> **Модификации коллекций**
>
> У **LazyCollection** нет метода `transform()`. Вместо него следует использовать `map()`.

Подобности о быстродействии коллекций читай ниже.

Только осмысленнные и необходимые. Их всегда следует вовремя актуализировать. И не только свои.

Комментариями можно описывать причины выбора текущего решения. Например, на данный момент нельзя реализовать иначе, потому что <...>, смотреть туда (ссылка на ТЗ, метод, класс, ...)

Также следует описывать любые неочевидные моменты, которые влияют на техничку или БП. Их лучше не держать в своей голове, а поделиться этим с коллегами. Например,

- "именно здесь мы не можем выполнять X раньше, чем Y, потому что по бизнесу это должно происходить вот так (ссылка)";
- "здесь находятся строки в формате X, потому что они уже поменяли свой формат ранее вон там (ссылка)".

Если переменных много, можно описать назначение каждой из них, но лучше их просто правильно назвать.

Всякая ерунда вроде "цикл по показателям", "увеличиваем счётчик" не только мешает, но и может вызывать новые вопросы.

TODO писать можно и нужно, если в участке кода есть задел на будущее и к нему необходимо скоро вернуться. Он должен быть убран после решения описанного. Если ты наткнулся на TODO-блок, то игнорировать его нежелательно.

Также см. phpdoc.

Обогащение данных

Данные можно брать напрямую из БД или других сервисов. При обработке относительно больших объёмов исходных данных, наращивать их дополнительными следует очень осторожно.

Не следует злоупотреблять сортировками, запросами к БД, многократными циклами, переборами массивов/коллекций, чтением/записью файлов, хранением контента файлов в ОЗУ...

Финализация классов

Делаем это редко, поскольку основная работа происходит в классах основных сущностях (читай ниже). Вполне допускается финализировать классы, которые не описывают бизнес-процессы.

Помни, что **финальные классы не мокаются в тестах!**

Цель бекенда – максимально снизить нагрузку на себя, вокруг себя и быстрее вернуть клиенту весь необходимый набор данных в наиболее полном виде.

Быстродействие коллекций Illuminate\Support\{Collection, LazyCollection}

Отбор данных: `where()` vs `filter()`

Например, нужно найти какой-то конкретный элемент коллекции по значениям его полей. Обычно для этого используют метод **`where()`** коллекций или его вариации (**`firstWhere()`**, **`whereIn()`** и т.п.).

Бенчмарки показывают, что все они в разы (а то и на порядки) медленнее, чем **`filter()`**.

Методы семейства **`where()`** разработчик вынужден вызывать каждый раз на каждое условие. При чейнинге, каждая новая отфильтрованная коллекция целиком итерирутся сызнова, вызывая внутри рекурсивный **`data_get()`** для парсинга ключей с точкой.

Метод **`filter()`** итерирует коллекцию лишь однажды, проверяя результат аргумента-замыкания. Оно должно вернуть булев тип, т. о. при итерации элементов можно описать сразу несколько условий.

Пример бенчмарка

```
// 2651
$data = (new NsiClient($jwt))
    ->collect('system/organisations/allowed');

Benchmark::dd([
    //    ,
    //    Collection / LazyCollection

    /* ~49.083ms / ~0.013ms */
    static fn () => $data
        ->where('org_type_id', 1),

    /* ~2.847ms / ~0.005ms */
    static fn () => $data
        ->filter(static fn ($row) => $row
['org_type_id'] === 1),

    /* ~50.883ms / ~0.017ms */
    static fn () => $data
        ->where('org_type_id', 1)
        ->where('org_structure_type_id', '!=',
null),
```

Ассоциация (индексирование): `keyBy(<string>)` vs `keyBy(<callable>)` vs `mapWithKeys(<callable>)`

Пусть существует некоторая коллекция, содержимое коей нужно ассоциировать по какому-то полю каждого из элементов, например, по id. Для того, чтобы в коллекции массив оказался ассоциированным, существуют эти два метода.

Метод **`keyBy()`** принимает только имя поля, которое нужно взять из элемента массива и сделать его ключом для этого элемента в массиве.

Метод **`mapWithKeys()`** принимает замыкание, которое должно вернуть ассоциированный массив с готовым ключом и его значением.

На практике, скорость **`mapWithKeys()`** во много раз выше, чем **`keyBy()`** .

Пример бенчмарка

```
// 2651
$data = (new NsiClient($jwt))
    ->collect('system/organisations/allowed');

Benchmark::dd([
    //    ,
    //    Collection / LazyCollection

    /* ~20.359ms / ~0.006ms */
    static fn () => $data->keyBy('id'),

    /* ~3.740ms / ~0.005ms */
    static fn () => $data->mapWithKeys(
        static fn ($row) => [$row['id'] => $row]
    ),
], 10);
```

Однако метод **`keyBy()`** также принимает на вход и замыкание, которое должно вернуть только значение ключа:

```
$data->keyBy(static fn ($row) => $row['id']));
```

Сортировка: `sortBy(<string>)` vs `sortBy(<callable>)`

Ситуация совершенно аналогична описанным слева. Если передать имя ключа строкой, то это будет сильно медленнее, чем передать сразу замыкание. И ленивые коллекции, вполне ожидаемо, на **порядки** быстрее.

Пример бенчмарка

```
// 2651
$collect = (new NsiClient($jwt))
    ->collect('system/organisations/allowed');

Benchmark::dd([
    //    ,
    //    Collection / LazyCollection

    /* ~23.509ms / ~0.007ms */
    static fn() => $data->sortBy('id'),

    /* ~4.220ms / ~0.008ms */
    static fn() => $data->sortBy(
        static fn ($row) => $row['id']
    ),
], 10);
```

```
    /* ~2.964ms / ~0.005ms */
    static fn () => $data
        ->filter(
            static fn ($row) => $row
['org_type_id'] === 1
                && $row['org_structure_type_id'] !
== null
        ),
], 10);
```

Данные по ключу: `pluck(<string>)` vs `map(<callable>)`

Коллекции могут возвращать все значения какого-то конкретного поля вложенных элементов. Проще всего вызвать **`pluck()`** с именем ключа, но, как ты догадываешься, **`map()`** с замыканием будет быстрее. Замыкание должно вернуть значение того поля из элемента, массив которых нужно получить на выходе:

Идентичный результат

```
$data->pluck('id');
$data->map(static fn ($row) => $row['id']));
```

Приведение к массиву: `toArray()` vs `all()`

Нередко бывает необходимость собрать список однотипных полей. Например, из всей коллекции объектов нам нужны только ID только активных записей:

```
$ids = $collection
    ->filter(static fn ($row) => $row->is_active)
    ->map(static fn ($row) => $row->id);
```

Мапа возвращает новую коллекцию, а нам нужен массив. Здесь у нас два варианта:

- `->all()` — вернёт всё содержимое коллекции как есть;
- `->toArray()` — тоже вернёт всё содержимое коллекции, но прежде — рекурсивно приведёт всё её содержимое к массивам, если есть нескалярные вложенные элементы.

Поскольку мы сформировали список интов, то нам не нужно бежать по нему ещё 1 раз, поэтому следует вызвать именно `all()`. Аналогично, если внутри нет никаких Arrayable объектов.

Советы по оптимизации обработки коллекций

1. В те методы коллекций, которые могут принимать замыкания, лучше передавать именно замыкания. Даже если это кажется глупым и многословным.
2. **`LazyCollection`** использует генераторы под капотом, поэтому для них п1 можно пренебречь (разница по быстродействию на уровне погрешности в наносекундах). Ленивые предпочтительнее обычных, но не имеют части функционала (например, сериализация).
3. Коллекции используются очень часто. В первую очередь, для получения данных из БД или НСИ. Для типизации значений внутри коллекций используется метод **`transform()`**. Если решено использовать **`LazyCollection`**, то **`transform()`** необходимо изменить на **`map()`**.
4. Да, обработка данных через генераторы быстрее, чем классическая обработка массивов. Но это не значит, что бекенд быстрее отдаст ответ клиенту. В общем случае, всеми переданными замыканиями (**`map(fn() => ..)`**, **`keyBy(fn() => ...)`**, ...) мы лишь откладываем фактическое выполнение операций над данными на этап сериализации (json-изации) данных, буквально перед ответом. Использование **`LazyCollection`** даёт преимущество в том, что *подготовка операций над данными* происходит быстрее. Однако перед ответом все или часть этих операции уже непосредственно выполняются в замыканиях, а поскольку количество данных одинаково, существенно скорость ответа бекенда не меняется.

Основные сущности

Model (модель)

Слой изменения, создания или удаления данных в БД. Непосредственно вызывает процедуры и функции на уровне БД (далее – **подпрограммы, ПП**).

1. Не допускается использование ORM и прямые insert/update/delete. Любые данные модифицируются только через ПП.
2. В моделях возможны следующие операции:
    a. старт транзакции – до непосредственного вызова ПП;
    b. вызов ПП, типизация результатов;
    c. фиксация транзакции (commit) – после успешного выполнения ПП;
    d. выброс исключения с откатом транзакции (rollback) – после НЕуспешного выполнения ПП.
3. Каждый метод класса модели может вызывать одну или несколько ПП подряд как того требует бизнес-сущность (артефакт).
4. Каждый метод модели должен выбрасывать исключение (Exception) в случае ошибки вызова ПП и откатывать всю транзакцию.
5. При успехе, метод модели должен зафиксировать транзакцию и вернуть данные от ПП либо **true**.
6. Данные, возвращаемые методом модели, следует строго типизировать.
7. Аргументы, которые в спецификации ПП объявлены как **in out** или **out**, должны передаваться по ссылке.

Как правило, модели **не наследуются** от Eloquent.

Также читай ниже о механизме ET.

View (представление)

Слой представления готовой информации потребителю (пользователю) сервиса. Не путать с представлениями в БД.

Фактически не используются. Фронтенд пишется на React (lis-web) отдельно от бекенда.

Бекенд-сервисы работают только в качестве API и возвращают ответы в формате JSON/XML.

Controller (контроллер)

Слой обработки входящих http-запросов.

1. Контроллеры принимают запрос, предварительно валидируют входящие параметры и отдают ответ. Между этими событиями обработка запроса может быть передана в сервис (о них ниже) и /или модель и/или репозиторий.
2. Следует внимательно относиться к этому слою: контроллер должен быть тонким.
3. Если какой-то БП требует больше одной операции (например, получить одни данные и изменить другие), то следует передать это сервису.
4. Если требуется только получить данные, то вызывается репозиторий и его результат отдаётся клиенту.
5. Если требуется только изменить какие-то данные, то следует вызвать модель и вернуть клиенту ответ об успехе.

Repository (репозиторий)

Слой чтения данных из БД.

- Не допускается использование ORM.
- Каждый метод класса репозитория должен вызывать одну ПП.
- Использование прямых DQL-запросов допустимо по согласованию с БД-разработчиками. Для **lis-arm** это только селекты из курсоров, возвращаемых ПП, а в **lis-nsi** кеширование справочников построено, в том числе, и на прямых селектах из таблиц.
- Каждый метод репозитория может выбрасывать исключение (DBException) в случае ошибки вызова ПП. Например, если данные необходимы по БП, но их не оказалось в БД.
- При успехе, метод репозитория должен вернуть данные, которые вернула ПП, или **true**.
- Данные, возвращаемые методом репозитория, следует строго типизировать.
- Аргументы, которые в спецификации ПП объявлены как **in out** или **out**, должны передаваться по ссылке.

Полезное можно почитать здесь: **Паттерн репозиторий**

Middleware

Слой промежуточной обработки входящих http-запросов и ответов.

Используется для проверки базовой информации на соответствие каким-то условиям, например, наличие обязательных заголовков, признака авторизации и пр. Может использоваться и в других целях, на усмотрение разработчика.

Service (сервис)

Слой бизнес-процессов.

В каждом сервисе реализуется целиком один БП. Сервис вызывается из контроллера после базовой проверки параметров входящего http-запроса.

В сервисах допускается всё, что необходимо для реализации БП: вызов репозиториев, моделей, обогащение данных, отправка http-запросов, работа с файлами и пр. В конечном счёте, сервис должен возвращать в контроллер готовые данные.

Если в рамках БП планируется изменение данных через 2+ модели, то необходимо использовать механизм единой транзакции БД (ET, о нём ниже). Таким образом, все потенциально изменяемые данные в БД будут защищены при возникновении ошибки где-нибудь в конце БП.

DTO (объект переноса данных)

Следует понимать буквально: служит только для передачи данных внутри приложения в строгих типах. Можно понимать как дженерик-тип.

Хранимые в DTO данные могут быть как скалярами и др. встроенными типами php, так и другими, неограниченно вложенными DTO и/или value objects.

Частый (и основной) сценарий использования – там, где метод класса имеет 4+ аргумента.

Value object (объект-значение)

Тоже следует понимать буквально. Это DTO, который участвует не только в переносе данных, но также *активно* участвует в БП.

VO может валидировать корректность значений не только технически (например, форматы дат), но и с точки зрения БП. Например, дата забора образца должна быть меньше даты внесения результатов исследований по этому образцу, однако это условие регулируется настройками системы и может быть в (ы)ключено глобально. Другой пример: проверка пароля на удовлетворение требованиям безопасности (количество символов, регистр, алфавит...). Такие вещи могут проверяться в VO.

Хранимые в VO данные могут быть как скалярами и др. встроенными типами php, так и другими, неограниченно вложенными DTO и/или VO.

Exceptions (исключения)

Следует выбрасывать везде, где есть рискованные процессы, особенно в БП.

Их сообщения :

1. должны быть осмысленными;
2. должны быть однозначными (не допускать разных трактовок);
3. должны быть по возможности краткими и уникальными, чтобы можно было быстро понять (или легко догадаться) на каком этапе БП она возникла;
4. должны быть описаны в файлах переводов;
5. могут иметь дополнительные сведения (например, значение, которое вызвало ошибку).

БД может сама вызывать исключения на своём уровне. Если оно возникло внутри БД, то бекенд его перехватит и сгенерирует ошибку уровня php. Такие ошибки чаще всего несут ненужную пользователю информацию, поэтому при необходимости лучше поработать в паре с БД-разработчиком либо перехватить исключение БД и сгенерировать новое вместо выброса готового.

Технические возможности и нюансы

Выполнение консольных команд

Команды (скрипты) **composer**, которые описаны в composer.json, следует выполнять на хосте, поскольку они нужны для управления самим проектом.

Команды **artisan** и другие следует выполнять изнутри контейнера, поскольку они нужны для работы с файлами проекта и Laravel:

Пример

```
docker exec -ti lis-arm php artisan route:list
docker exec lis-arm ./vendor/bin/phpunit
```

Анализ кода и pre-commit git hook

Хук выполняет операции анализа кода синтаксического, статического и его тестирования.

Устанавливается автоматически после установки зависимостей. Можно установить вручную командой **`composer git-hooks-install`**.

Хук не позволит выполнить **git commit**, если код не проходит проверки код-стайла, стат. анализом и тестированием.

Выполнять эти операции – ответственность и совесть разработчика. Заботимся о себе и коллегах, не позволяем грязи испортить кодовую базу и в перспективе усложнить её поддержку.

Batch-запросы (lis-arm)

Что это такое – читай здесь: https://www.jsonrpc.org/specification#batch

В объекте класса `\Illuminate\Http\Request` есть флаг **`is_batch`**. Он равен true если в API пришёл batch-запрос, иначе false.

В конце обработки batch-запроса единая транзакция моделей (о ней ниже) коммитится.

Сессионные переменные

Каждый запрос к API должен сопровождаться заголовком **`Authorization: Bearer <access_token>`**, где <access_token> есть JWT, выданный Keycloak-ом. Он содержит полезную нагрузку.

Эта полезная нагрузка доступна в общем контейнере приложения через **`app('jwt')`**.

Например, можно получить доступ к ID организации пользователя, в которой он сейчас находится: **`app('jwt')->org_id`**

Полный набор этих данных можно узнать, изучив документацию lis-core или вставив токен в JWT.io.

Artisan и нюансы кэширования конфигурации

Так как в API используется динамическая маршрутизация, то команда **`php artisan route:cache`** зафиксирует все маршруты, статически используемые в системе, и таблица маршрутов будет пуста для каждого запроса в API. Всвязи с этим, не рекомендуется выполнять эту команду.

Однако, при использовании команды **`php artisan optimize`** (оптимизация фреймворка, кэширование всех конфигурационных файлов, сжатие кода контроллеров и моделей в один файл кэша), следует после нее выполнять **`php artisan route:clear`**.

> Так же при ЛЮБОМ изменении конфигурационного файла (параметры, ключи) нужно использовать **`php artisan config:clear`**

Работа с НСИ (lis-nsi)

Очень много данных в LisUP хранится в кешах. Работу с этими данными на себя взял отдельный сервис lis-nsi. Он может отдавать данные из кешей и обновлять данные в кешах.

В некоторых проектах существует `App\Extensions\LisNsiClient\NsiClient`. Это специальный HTTP-клиент, который предоставляет интерфейс взаимодействия с lis-nsi по обновлению и получнеию данных. Доступ к нему осуществляется через фасад `App\Extensions\LisNsiClient\NsiFacade`.

Чтобы получить данные, нужно:

Обрати внимание

Работа со справочниками раскрыта в этих разделах:

- Система кеширования данных (НСИ)
- Уголок Back-End разработчика lis-nsi

- (у)знать UNIT_CODE таблицы, из которой следует забрать данные (для забора "сырых" табличных данных)

```
$data = NsiFacade::fromUnitCode('RB_RESEARCH_GROUP')->onlyOrgId($params['org_id'])->collect();
```

- или (у)знать URI метода, который предоставляет такую возможность (для забора специфичных и/или обогащённых данных)

```
$report_params = NsiFacade::fromPath('system/reports/params')->collect();
$report_params = NsiFacade::collect('system/reports/params');
```

Для фильтрации записей по полю **org_id** следует вызвать **filter(['org_id' => $org_id])** или **onlyOrgId($org_id)**.

Для фильтрации записей по их **ID** следует вызвать **filter(['id' => $some_ids])** или **onlyIds($some_ids).**

Для фильтрации записей по другим полям следует вызвать **filter(['somefield' => $somevalue])** (где **$somevalue** есть скаляр или массив скаляров).

Для получения списка только указанных полей следует вызвать **fields('id', 'code')** (допускается передача списка в виде массива).

Для получения данных следует вызывать метод **collect()**. Этот метод принимает так же URI метода lis-nsi API.

Клиент по умолчанию возвращает коллекцию **Illuminate\Support\Collection**. Для получения генератора **LazyCollection** следует перед вызовом **collect()** вызвать сеттер **lazy()**.

Метод **collect()** также принимает вторым аргументом массив параметров, которые будут переданы URL запроса.

Для обновления данных в кешах следует вызывать методы **add()**, **update()** или **delete()**.

Более подробно изучить работу клиента НСИ можно по исходникам.

Работа с БД

Общее

Для доступа к базе данных **не используются** ORM или Eloquent. **Следует использовать несколько команд через расширенный фасад \App\Extensions\DB\DB.**

Они подробно описаны ниже. Каждый из них принимает вторым аргументом массив с привязками переменных (bindings). Внутри могут быть как скаляры, так и массивы скаляров.

Вот пример всех возможных вариантов составления этих привязок:

```
DB::select('select something from somewhere', [
    'pnLPU' => $lpu, //
    'pnTYPE' => MyEnum::SomeValue, //  BackedEnum
    'psTEMPLATE_CODE' => [ //
        'value' => $params['code'],
        'type' => PDO::PARAM_STR,
    ],
    'pnTEMPLATE_ID' => [ //
        'value' => $template_id,
        'type' => PDO::PARAM_INT,
```

```
    ],
    'pnID' => [ //  in out  /
        'value' => &$id, //  $id    ,
        'type' => PDO::PARAM_INPUT_OUTPUT,
        'length' => 17, //
    ],
    'pcDS_IDS' => [ //
        'value' => [
            1234567890,
            9012345678,
            3216549871,
        ],
        'type' => 'CL_ID', // 'D_CL_ID  lisup 1.5
    ],
    'pcCODES' => [ //
        'value' => [
            'lorem-ipsum',
            'dolor-sit-amet',
        ],
        'type' => 'CL_STR', // 'D_CL_STR  lisup 1.5
    ],
]);
```

Выборка данных

DB::select($sql, $bindings)

Позволяет сделать прямую выборку из таблиц, представлений, курсоров и ПП по сырому sql-запросу. Пример вызова указан выше.

DB::getCursor($sql, $bindings, $useBroker)

Полностью дублирует функционал **`DB::select()`**, однако выгодно отличается дополнительным функционалом.

Метод умеет получать данные из рекурсивных курсоров (вложенных друг в друга). Такие курсоры могут возвращать некоторые ПП БД.

Также, этот метод умеет работать с брокером и пагинировать данные.

Дополнительными параметрами являются переменные пагинации и сортировки:

- **`cstart_, cend_`** - используются совместно и указывают на диапазон выбираемых данных.
  При указании [1, 100] будет возвращено 100 записей с просчетом пагинационных переменных в общей выборке.

- **`csort_`** - используется для сортировки по полю выборки и задаётся строкой в формате "<><>< >" (например, `'pat_fio asc'`)
- **`cfilter_`** - используется для фильтрации выборки и задаётся картой (ассоц. массивом) в формате `["<1>" => <1>, ...]`

При использовании **`DB::getCursor()`** они неявно включаются в общий массив биндингов из пришедшего http-запроса.

`:     ,    - -      Postgresql`

Ниже представлены примеры вызова метода:

Пример выборки без брокера

Пример вызова брокера

```
DB::getCursor(
    'select D_PKG_DRIVER_MANAGER_API.GET_DEVICES_SAMPLE(:psDEVICES) from dual',
    ['psDEVICES' => implode(';', $device_uuids)],
    false
);
```

Все переданные параметры привяжутся как есть напрямую.

```
DB::getCursor(
    'select D_PKG_LABMED_ORDER_API.GET_LIST_FOR_CHANGE(
                pclParams => :pclParams,
                pclValues => :pclValues
            )
       from dual',
    [
        "pnLPU" => $this->lpu_id,
        "psSAMPLE_CODE" => $sample_code,
        "psDIRECTION" => $dir_numb,
        "psFIO" => $agent_fio
    ]
);
```

Все ключи переданных параметров привяжутся в **pclParams**, а значения в **pclValues.**

⚠ **Про типы данных**

Что нужно учитывать при типизации. Данные от БД могут быть получены в одном из трёх нативных типов php:

- ресурс (фактически, курсор) - все операции по fetch'у курсоров берет на себя фасад DB;
- null - нал есть нал, значит ничто;
- строка - буквально всё остальное: числа целые, дробные, строки, даты...

При получении данных почти всегда следует их паковать в коллекцию и полноценно протипизировать (числа в числа, варианты в BackedEnum, даты в Carbon и т.д.):

Пример вызова брокера

```
$data = collect(DB::getCursor(...))
    ->transform(static fn ($row) => [
       'field1' => (int)$row['field1'],
       'date' => Carbon::parse($row['date']),
       'status' => StatusEnum::from($row['status']),
       ...
    ]);
```

Изменение данных

DB::executeFunction()

Позволяет выполнить функцию БД и получить значение, которое она вернула:

Пример вызова

```
DB::executeFunction(
    //   F_GEN_NEXT_VALUE  D_PKG_LBM_BARCODE_SETTINGS
    'D_PKG_LBM_BARCODE_SETTINGS.F_GEN_NEXT_VALUE',
```

```
    //
    [
        'pnOBJECT_TYPE' => $params['object_type'],
        'pnLPU' => $params['lpu'] ?? auth()->user()->lpu,
        'pnDIR_ID' => $params['dir_id']
    ],
    PDO::PARAM_STR, //
    20              //  .
);
```

DB::executeProcedure()

Позволяет выполнить функцию и не ожидать результата:

Пример вызова 1

```
$result = DB::executeProcedure(
    //   TAKE_IN  D_PKG_LABMED_SAMPLES
    'D_PKG_LABMED_SAMPLES.TAKE_IN',

    //
    [
        'pnID'  => $params['id'],
        'pnLPU' => $params['lpu'] ?? auth()->user()->lpu,
        'pnCROCKERY_PREF' => $params['crockery_pref']
    ]
);
```

При успехе **`$result`** может быть равен только **`true`**. Больше ничего не вернётся. При ошибке выстрелит исключение.

Для **out** или **in out** аргументов процедуры следует передавать переменные по ссылке и указывать её длину:

Пример вызова 2

```
$result = DB::executeProcedure(
    'D_PKG_LABMED_ORDER_API.ADD_AGENT',
    [
        'pnAGENT_ID' => [
            'value' => &$id,
            'type' => PDO::PARAM_INPUT_OUTPUT,
            'length' => 17,
        ],
        'pnLPU' => $lpu_id,
        // ...
    ]
);
return (int)$id;
```

При успехе **`$result`** может быть равен только **`true`**, а $id = числовой строке. При ошибке выстрелит исключение.

Управление транзакциями

`DB::beginTransaction()` - старт транзакции.

`DB::commit()` - приём транзакции.

`DB::rollback()` - откат транзакции.

`DB::transactionLevel()` - узнать уровень транзакции. Если === 0, значит открытых транзакций нет.

Дополнительные команды по модели Eloquent можно посмотреть в документации Laravel: https://laravel.com/docs/database

Рекомендуется использовать механизм единой транзакции. О нём ниже.

Механизм единой транзакции для моделей

ET всегда начинается только при явном вызове, но её необязательно вручную завершать. Она закоммитится автоматически после обработки входящего http-запроса.

Доступен в некоторых сервисах, работающих с БД напрямую.

Позволяет заключить **все** модели в одну транзакцию, что избавляет от неполноты данных и их повреждения. При этом очень прост в реализации и использовании и не мешает моделям работать самостоятельно.

Простой пример. В процессе сохранения результатов показателей исследований, происходит множество действий: проверка и сохранение введённых вручную значений, расчёт вычислимых (aka расчётных aka расчитываемых) показателей, валидация результатов (отдельный сложный БП) и прочее. Допустим, сохранение введённых вручную результатов прошло успешно, но при автоматическом расчёте других показателей возникла фатальная ошибка.

Механизм единой транзакции позволяет откатить всю транзакцию целиком, чтобы в БД не сели уже готовые результаты -- то есть откатить весь БП.

Такое можно (на самом деле, даже нужно) проворачивать и при batch-запросах: далеко не всегда возможно уложить весь БП в один метод API.

Единая транзакция коммитится автоматически в конца обработки http-запроса и откатывается (разработчик обязан об этом позаботиться) при возникновении исключений.

Как с ней обращаться:

1. Для инициализации общей транзакции следует перед вызовом моделей (в контроллерах, сервисах) вызвать **`BaseModel::setSingleTransaction()`**.
   В этом случае глубина вложенности транзакций **`DB::transactionLevel()`** всегда будет **`=== 1`** и все модели, унаследованные от BaseModel, будут отрабатывать в рамках неё.

2. Для ручного завершения общей транзакции следует вызвать метод **`BaseModel::setSingleTransaction(false)`**.
   В этом случае, если общая транзакция начата, то произойдёт коммит всех изменений.

3. В моделях-наследниках:
   a. Обернуть вызовы ПП в **try-catch**
   b. До блока **try** (вызова ПП) добавить вызов **`self::beforeExecute()`**.
      Если есть **`DB::beginTransaction()`** - заменить.

   c. В конце блока **try** (после вызова ПП) добавить вызов **`self::afterExecute()`**.
      Если есть **`DB::commit()`** - заменить.

   d. Внутри блока **catch до выброса исключений** добавить (оставить) вызов **`DB::rollback()`**
      Таким образом достигается откат ET на любом шаге БП, чтобы вовремя выстрелить ошибкой в клиента через весь стек вызовов.

Ниже представлены сухие участки кода на примере ЕТ в рамках БП возврата результатов исследований на рассмотрение:

JourspController

```
public function returnToReview(RequestInterface
$request): array
{
    // ...
    $service = new JourspReviewService(/*...*/);
    // ...
    return [
        'result' => $service->returnToReview(/*...*
/)
    ];
}
```

JourspReviewService

```
public function returnToReview(): array
{
    //
    BaseModel::setSingleTransaction();

    $result = [];
    foreach ($this->params['joursps'] as $joursp) {
        $result[$joursp['id']] = $this->model-
>returnToReview(...);
    }

    // ...
    return $result;
}
```

JourspModel

```
public function returnToReview(/*...*/): bool
{
    self::beforeExecute();
    //   --^    ,      returnToReview()

    try {
        DB::executeProcedure(/*...*/);

        self::afterExecute();
        //  --^      ,

        return true;
    } catch (Exception $e) {
        DB::rollback();
        throw new UniException($e->getMessage());
        //    ,
    }
}
```