Лабораторная работа №6
«Сервисы»

Цель работы

Познакомиться с практической реализацией принципа инверсии зависимостей, принципа единственной ответственности и механизмом внедрения зависимостей

Задание для выполнения

Реализуйте сервисы на уровне приложения, реализующие с доменными моделями те же действия, что и сценарий транзакции из ЛР №1. Не забудьте о принципе единственной ответственности.

Хотя бы одно действие должно реализовываться доменным сервисом.

Напишите модульные тесты для сервисов. Поскольку инфраструктурный слой еще не реализован – используйте заглушки вместо репозиториев (ЛР №4).

Теоретические сведения

Вспомните, что изучили на предыдущих двух работах.

Говоря в терминах многослойной архитектуры, один из общих подходов к реализации бизнес-логики состоит в расщеплении слоя предметной области на два самостоятельных слоя: «поверх» модели предметной области располагается слой служб. Логика слоя представления взаимодействует с бизнес-логикой исключительно при посредничестве **слоя служб**, который действует как API приложения.

Поддерживая внятный интерфейс приложения (API), слой служб подходит также для размещения логики *управления транзакциями* и *обеспечения безопасности*. Это даёт возможность снабдить подобными характеристиками каждый метод слоя служб.

Основное решение, принимаемое при проектировании слоя служб, состоит в том, какую часть функций уместно передать в его ведение. Самый скромный вариант – представить слой служб в виде промежуточного интерфейса, который только и делает, что направляет адресуемые ему вызовы к нижележащим объектам. В такой ситуации слой служб обеспечивает API, ориентированный на определённые *варианты использования* приложения, и предоставляет удачную возможность включить в код функции-оболочки, ответственные за управление транзакциями и проверку безопасности.

Другая крайность – в рамках слоя служб представить большую часть логики в виде *сценариев транзакции*. Нижележащие объекты домена в этом случае могут быть тривиальными; если они сосредоточены в модели предметной области, удастся обеспечить их однозначное отображение на элементы базы данных и воспользоваться более простым вариантом слоя источника данных (скажем, активной записью).

Между двумя указанными полюсами существует вариант, представляющий собой более, нежели смесь двух подходов: речь идёт о модели *контроллер-сущность*. Главная особенность модели заключается в том, что логика, относящаяся к отдельным транзакциям или вариантам использования, располагается в соответствующих сценариях транзакции, которые в данном случае называют контроллерами (или службами) [1].

В гексагональной архитектуре, которая принята за основу для построения ваших лабораторных работ, роль слоя служб играет **уровень приложения**:

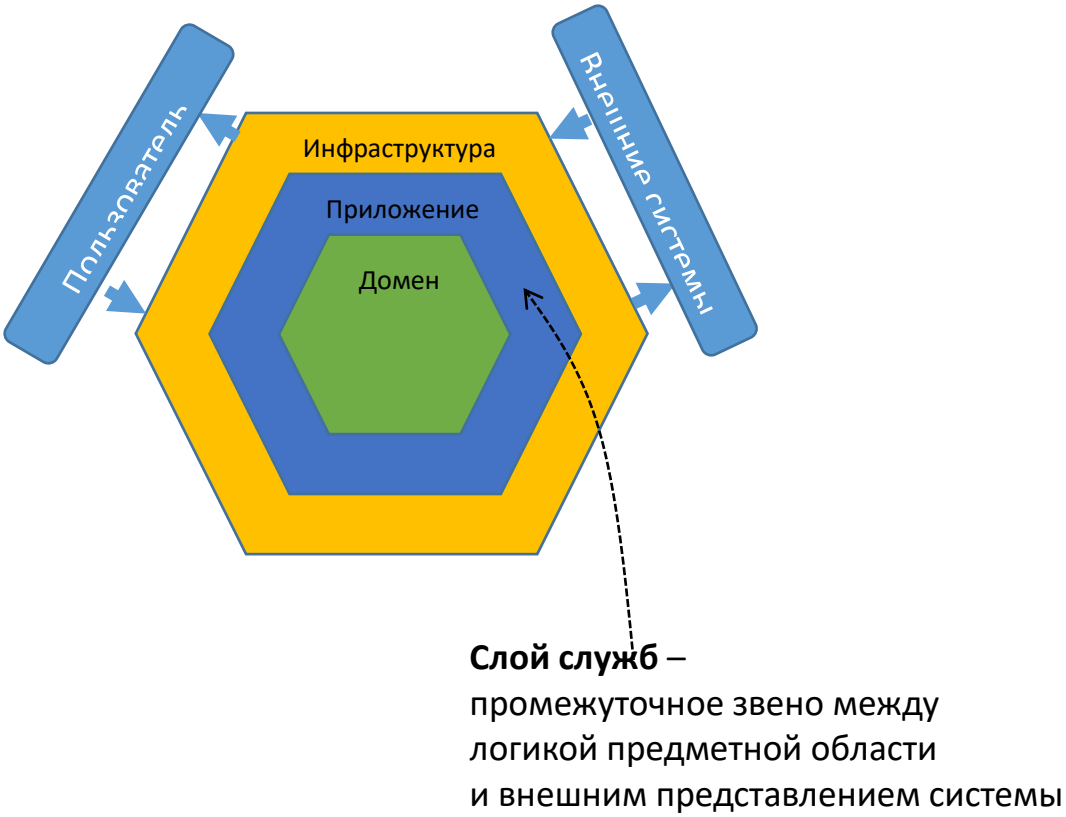

Рисунок 1 – Место слоя служб в гексагональной архитектуре

Уровень приложения – это слой, который отделяет доменную модель от клиентов, которые запрашивают или изменяют ее состояние. Службы приложения – это строительные блоки для подобного слоя. Согласно Вон Вернону: «Службы приложения – это непосредственные клиенты доменной модели». Служба приложения – точка контакта между внешним миром (HTML формы, клиенты API, фреймворки, UI, CLI и т.д.) и самой Доменной моделью. Это верхний уровень вариантов использования, которые ваша система

предоставляют миру: «как гость, я хочу зарегистрироваться», «как авторизованный пользователь, я хочу купить товар», и т.д.

Службы (сервисы)

Отдельно стоит отметить, что наряду со службами уровня приложения гексагональная архитектура и DDD подразумевают возможность существования ещё других служб.

Согласно Эрику Эвансу, «*когда значительный процесс трансформации в домене не является естественной ответственностью Сущности или Объекта значений, добавьте операцию к модели как отдельно стоящий интерфейс, объявленный как Сервис. Определите интерфейс в терминах языка модели и удостоверьтесь, что наименование операции является частью Единого Языка. Сделайте Сервис без сохранения состояния*».

Так что, когда необходимо представить операцию, для которой сущности и объекты значений не лучшее место, нужно моделировать её как сервис. В DDD есть три типа сервисов:

- *сервисы приложения*: оперируют скалярными типами, трансформируя их в типы предметной области. Скалярными являются любые типы, неизвестные Доменной модели. Сюда включаются примитивные типы, и типы, не относящиеся к домену;
- *доменные сервисы*: оперируют только типами, принадлежащими домену. Они содержат значимые концепции, которые могут быть найдены в Едином языке. Реализуют операции, которые недостаточно хорошо подходят сущностям или объектам значений;
- *инфраструктурные сервисы*: все операции, которые реализуют касаются инфраструктуры, такие как отправка почты или журналирование значимых данных. В терминах гексагональной архитектуры, они лежат вне границ домена [2].

Сервисы и инверсия зависимостей

Рассмотрим пример объявления и реализации нескольких сервисов разных уровней.

Микросервис **Stripe** отвечает за интеграцию системы с внешним платёжным агрегатором https://stripe.com с использованием его API. Рассмотрим один из вариантов использования – занесение денег (*charge*) на счет продавца, т.е. непосредственно факт оплаты с подготовленного заранее источника (*source*, например, кредитной карты покупателя).

Доменная модель оплаты `Payment` учитывает статус прохождения платежа по всем этапам, и, в том числе, позволяет выполнить действие `charge()`. При этом действия по взаимодействию с внешней системой – обмен данными по API Stripe – одновременно являются и частью предметной области для данного ограниченного контекста, и должны реализовываться на инфраструктурном уровне, поскольку работают с внешним миром. На помощь

приходит принцип инверсии зависимостей, поскольку доменная модель `Payment` не должна зависеть от инфраструктурной реализации:

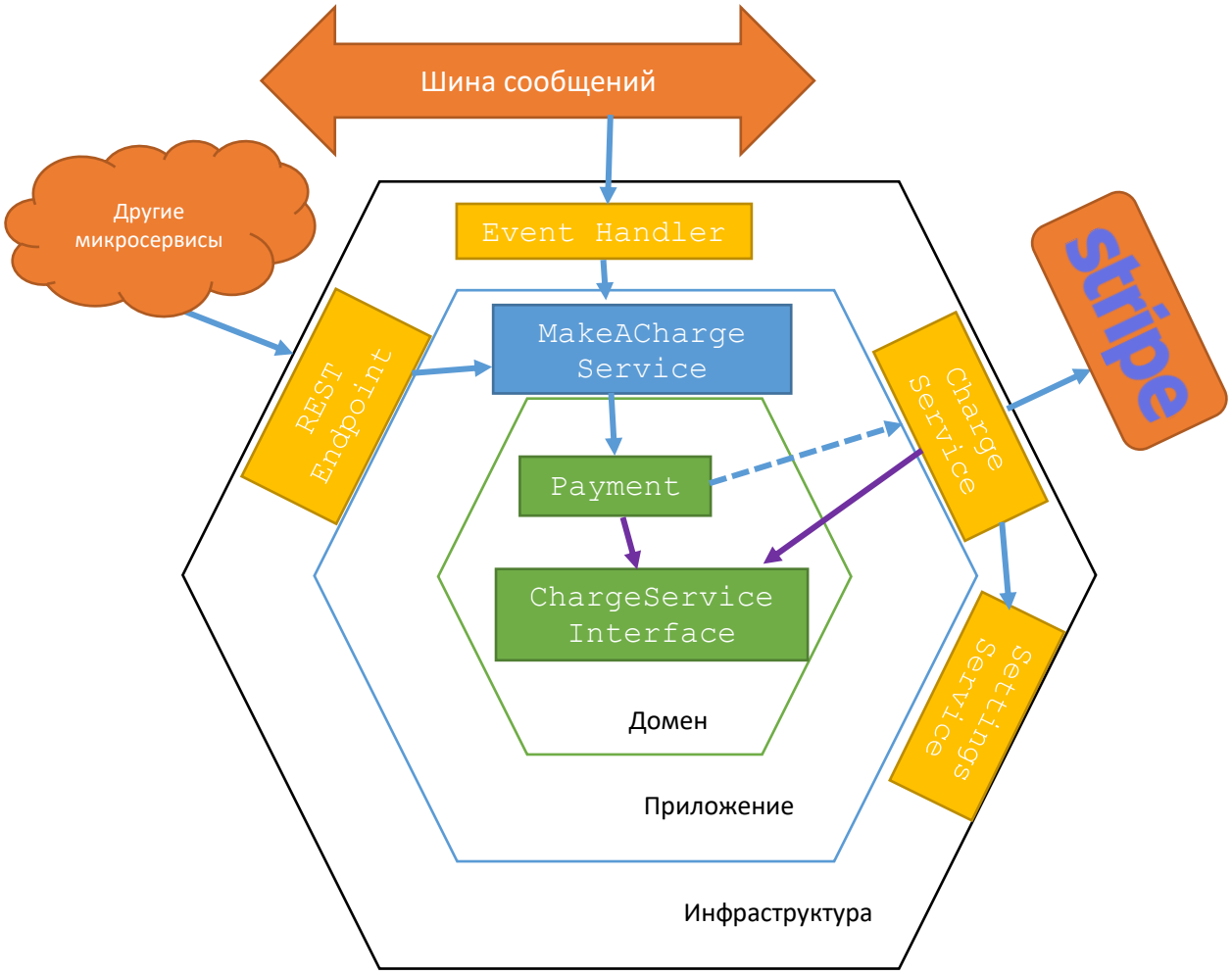

Рисунок 2 – Схема потоков управления и зависимостей между уровнями гексагональной архитектуры микросервиса Stripe

На Рисунке 2 изображено два варианта входа в вариант использования `MakeACharge`: по событию из шины сообщений (например, в другом микросервисе **Finance** произошло событие `ReadyToCharge`), или напрямую по REST API будет произведен запрос от другого микросервиса.

В обоих случаях на инфраструктурном уровне в соответствующих обработчиках формируется DTO для передачи на уровень приложения и вызова API домена – сервиса приложения `MakeAChargeService`.

```php
namespace My\Stripe\Infrastructure\EventHandler\ReadyToCharge;

use My\ApiCore\Infrastructure\Domain\Event\EventHandlerInterface;
use My\ApiCore\Service\Finance\Event\ReadyToCharge;
use My\Stripe\Application\MakeACharge\MakeAChargeCommand;
use My\Stripe\Application\MakeACharge\MakeAChargeService;
use Psr\Logger\LoggerInterface;
```

```php
class ChargeSource implements EventHandlerInterface
{

    /**
     * @var MakeAChargeService
     */
    private $makeAChargeService;

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

    public function __construct(
        MakeAChargeService $makeAChargeService,
        LoggerInterface $logger
    )
    {
        $this->makeAChargeHandler = $makeAChargeHandler;
        $this->logger = $logger;
    }

    /**
     * @param ReadyToCharge $event
     */
    public function handle($event)
    {
        try {
            $this->makeAChargeService->handle(
                new MakeAChargeCommand($event->getId())
            );
        } catch (\Exception $ex) {
            $this->logger->alert($ex);
        }
    }
}
```

Как видите, на инфраструктурном уровне необходимо обработать исключения, которые могут произойти на внутренних уровнях. Почему? Потому что это точка входа в приложение, верхний уровень обработчика, и дальше по стеку вызовов исключение передавать попросту некому – там только несколько уровней вызовов фреймворка, а он за логику приложения не отвечает. Следовательно, если вы хотите иметь какую-то информацию о произошедшей исключительной ситуации, или как-то её обработать – обработка должна быть произведена здесь. Например, можно записать в лог, как в примере, или бросить событие о неудачной обработке в шину.

На уровне приложения сервис `MakeAChargeService` является точкой входа в доменную модель для реализации операции зачисления средств продавцу. Обратите внимание на зависимости:

```php
namespace My\Stripe\Application\MakeACharge;

use My\Ddd\Application\Service\TransactionInterface;
```

```php
use My\Ddd\Domain\Event\EventDispatcherInterface;
use My\Stripe\Domain\Model\Buyer\BuyerRepositoryInterface;
use My\Stripe\Domain\Model\Payment\PaymentId;
use My\Stripe\Domain\Model\Payment\PaymentRepositoryInterface;
use My\Stripe\Domain\Model\Seller\SellerRepositoryInterface;
use My\Stripe\Domain\Model\Source\SourceRepositoryInterface;
use My\Stripe\Domain\Service\Charge\ChargeServiceInterface;

class MakeAChargeService
{
    /**
     * @var EventDispatcherInterface
     */
    protected $eventDispatcher;
    /**
     * @var TransactionInterface
     */
    private $transaction;

    /**
     * @var PaymentRepositoryInterface
     */
    private $paymentRepository;
    /**
     * @var SourceRepositoryInterface
     */
    private $sourceRepository;
    /**
     * @var SellerRepositoryInterface
     */
    private $sellerRepository;

    /**
     * @var ChargeServiceInterface
     */
    private $chargeService;

    public function __construct(
        PaymentRepositoryInterface $paymentRepository,
        SourceRepositoryInterface $sourceRepository,
        BuyerRepositoryInterface $buyerRepository,
        SellerRepositoryInterface $sellerRepository,
        TransactionInterface $transaction,
        ChargeServiceInterface $chargeService,
        EventDispatcherInterface $eventDispatcher
    )
    {
        $this->paymentRepository = $paymentRepository;
        $this->sourceRepository = $sourceRepository;
        $this->transaction = $transaction;
        $this->chargeService = $chargeService;
        $this->buyerRepository = $buyerRepository;
        $this->sellerRepository = $sellerRepository;
        $this->eventDispatcher = $eventDispatcher;
    }

    public function handle(MakeAChargeCommand $command): MakeAChargeResult
    {
        $payment = $this->paymentRepository->get(new PaymentId($command-
>getPaymentId()));
        $source = $this->sourceRepository->get($payment->getSourceId());
        $buyer = $this->buyerRepository->get($source->getBuyerId());
```

```
        $seller = $this->sellerRepository->get($source->getSellerId());

        $payment->charge(
            $source,
            $buyer,
            $seller,
            $this->chargeService
        );

        $this->transaction->execute(function () use ($payment) {
            $this->paymentRepository->save($payment);
            $this->eventDispatcher->dispatch($payment->releaseEvents());
        });

        return new MakeAChargeResult(true);
    }
}
```

Как видим, с уровня приложения зависимости ведут только на уровень домена *(My\Ddd* - модуль, подключенный в данном и прочих микросервисах при помощи composer, является микрофреймворком для DDD; аналогично во все микросервисы подкючен My\ApiCore*, в котором поддерживаются контракты микросервисов, будет рассмотрено на следующих работах)*:

```
use My\Stripe\Domain\Model\Buyer\BuyerRepositoryInterface;
use My\Stripe\Domain\Model\Payment\PaymentId;
use My\Stripe\Domain\Model\Payment\PaymentRepositoryInterface;
use My\Stripe\Domain\Model\Seller\SellerRepositoryInterface;
use My\Stripe\Domain\Model\Source\SourceRepositoryInterface;
use My\Stripe\Domain\Service\Charge\ChargeServiceInterface;
```

При том, что интерфейсы репозиториев объявлены в доменной модели, их реализации относятся к инфраструктурному уровню – DIP.

Как видим, в вызов `$payment->charge(…)` передаются уже загруженные доменные модели, необходимые для проведения логики платежа, а также доменный сервис `ChargeServiceInterface`. На Рисунке 2 видно, что при $payment должен выполнить данную операцию при помощи `ChargeService`, который реализован на инфраструктурном уровне – штриховая стрелка показывает направление потока управления. Однако это направление не соответствует требованиям гексагональной архитектуры (и принципов DIP и SDP), поэтому и вводится `ChargeServiceInterface`, позволяющий инвертировать зависимость (фиолетовые стрелки).

```
namespace My\Stripe\Domain\Model\Payment;

use My\Ddd\Domain\Contract\AggregateRootInterface;
use My\Ddd\Domain\Traits\AggregateTrait;
use My\Stripe\Domain\Model\Buyer\Buyer;
use My\Stripe\Domain\Model\Seller\Seller;
```

```php
use My\Stripe\Domain\Model\Source\Source;
use My\Stripe\Domain\Service\Charge\ChargeServiceInterface;

class Payment implements AggregateRootInterface
{
    use AggregateTrait;

    // ...

    public function charge(
        Source $source,
        Buyer $buyer,
        Seller $seller,
        ChargeServiceInterface $charger
    ) {
        // ... key business logic preparing charge

        $chargeResult = $charger->execute(
            $source->getStripeId(),
            $seller->getStripeAccountId(),
            $buyer->getStripeCustomerId($seller),
            $this->getOrderId(),
            $this->getTotal()->getAmount()->getValue(),
            $this->getTotal()->getCurrency()->getValue()
        );

        if (!$chargeResult->isSuccess()) {
            $this->fail(
                 $chargeResult->getFailureCode(),
                 $chargeResult->getFailureMessage()
            );
            return;
        }

        $this->paid();
    }

    // ...
}
```

Сервису `ChargeService` на инфраструктурный уровень передаются данные, необходимые для вызова API Stripe, но не доменные модели. Вместо того, чтобы передавать пять скалярных параметров, можно было бы сформировать DTO – однако это уже скорее дело вкуса.

Доменный сервис – по DDD это интерфейс:

```php
namespace My\Stripe\Domain\Service\Charge;

interface ChargeServiceInterface
{
    public function execute(
        string $stripeSourceId,
        string $stripeAccountId,
        string $stripeCustomerId,
        string $orderId,
        int $amount,
        string $currency
```

```
    ): ChargeResult;
}
```

Реализация данного сервиса – на инфраструктурном уровне:

```
namespace My\Stripe\Infrastructure\Domain\Service\Charge;

use My\Stripe\Domain\Service\Charge\ChargeResult;
use My\Stripe\Domain\Service\Charge\ChargeServiceInterface;
use My\Stripe\Infrastructure\Communication\SettingsService;
use Psr\Logger\LoggerInterface;
use Stripe;

class RestChargeService implements ChargeServiceInterface
{
    /**
     * @var SettingsService
     */
    private $settingsService;
    /**
     * @var LoggerInterface
     */
    private $logger;

    public function __construct(
        SettingsService $settingsService,
        LoggerInterface $logger
    ) {
        $this->settingsService = $settingsService;
        $this->logger = $logger;
    }

    public function execute(
        string $stripeSourceId,
        string $stripeAccountId,
        string $stripeCustomerId,
        string $orderId,
        int $amount,
        string $currency
    ): ChargeResult
    {
        Stripe\Stripe::setApiKey($this->settingsService->getApiKey());

        try {
            $charge = Stripe\Charge::create(
                $this->prepareChargeData(
                    $amount, $currency, $orderId,
                    $stripeSourceId, $stripeCustomerId
                ),
               ['stripe_account' => $stripeAccountId]
            );
        } catch (\Exception $ex) {
            $this->logger->alert($ex));

            return ChargeResult::fail($ex->getStripeCode(), $ex->getMessage());
        }

        return ChargeResult::success($charge);
    }
```

```
}
```

Как видите, в данном сервисе есть ещё одна зависимость – SettingsService, который расположен также на инфраструктурном уровне и отвечает за настройки взаимодействия с API Stripe, например, за хранение и получение ApiKey. Почему же этот сервис передается в конструктор напрямую, а не через интерфейс? Потому, что мы и так находимся на инфраструктурном уровне и инвертировать зависимость от другого сервиса инфраструктурного уровня нет необходимости.

Возврат значений и синхронность выполнения

В рассмотренном примере все вызовы, начиная с точки входа и далее, являются синхронными. Это значит, что, переходя границы уровней в одном направлении, в противоположном мы можем возвращать некий результат действий.

Асинхронная обработка события

Очевидно, что это нужно не всегда. Например, если мы обрабатываем событие из шины сообщений, то брокер получает подтверждение о том, что его сообщение получено и начата обработка сразу, не дожидаясь результата обработки. В этом случае при необходимости возврата некоего ответа по результатам обработки можно бросить в шину другое событие с результатом, которое также асинхронно будет обработано в другом заинтересованном в этом результате микросервисе.

По схожей модели может быть построено и взаимодействие с пользовательским интерфейсом в браузере клиента: REST-запрос на сервер может только запускать процесс обработки, а ответ асинхронно придёт клиенту в WebSocket.

Синхронный вызов REST или gRPC

С другой стороны, если микросервис предоставляет REST API, или вызовы по gRPC, то может сразу же возвращать результат обработки. Тогда синхронность выполнения на всех уровнях только на руку.

Шина обработки команд

Однако, такая реализация не является до конца идеологически верной. В современных системах переход границы между инфраструктурным уровнем и уровнем приложения (т.е. границы между внешним представлением и бизнес-логикой) часто делается с использованием шины команд, которая умеет по типу переданной в шину команды (DTO `MakeAChargeCommand` в нашем примере) определять, какой обработчик на уровне приложения должен быть вызван. Это, во-первых, ещё более снижает связность уровней; а, во-вторых,

позволяет синхронную или асинхронную обработку команд в зависимости от реализации `CommandBus`.

CQRS

Следующим этапом в развитии архитектуры данного сервиса может быть применение шаблона CQRS – Command-Query Responsibility Segregation.

В двух словах его смысл в следующем: есть команды, которые изменяют состояние сервиса, но не возвращают результата; а есть запросы, которые не меняют состояние, но возвращают данные.

Для команд, по сути, ничего не меняется – весь процесс полностью соответствует представленному в примере (ну или с `CommandBus`).

Для запросов же есть одно важное отличие – поскольку они только возвращают данные, но не производят полезных действий в бизнес-логике сервиса, то домен они и не используют!

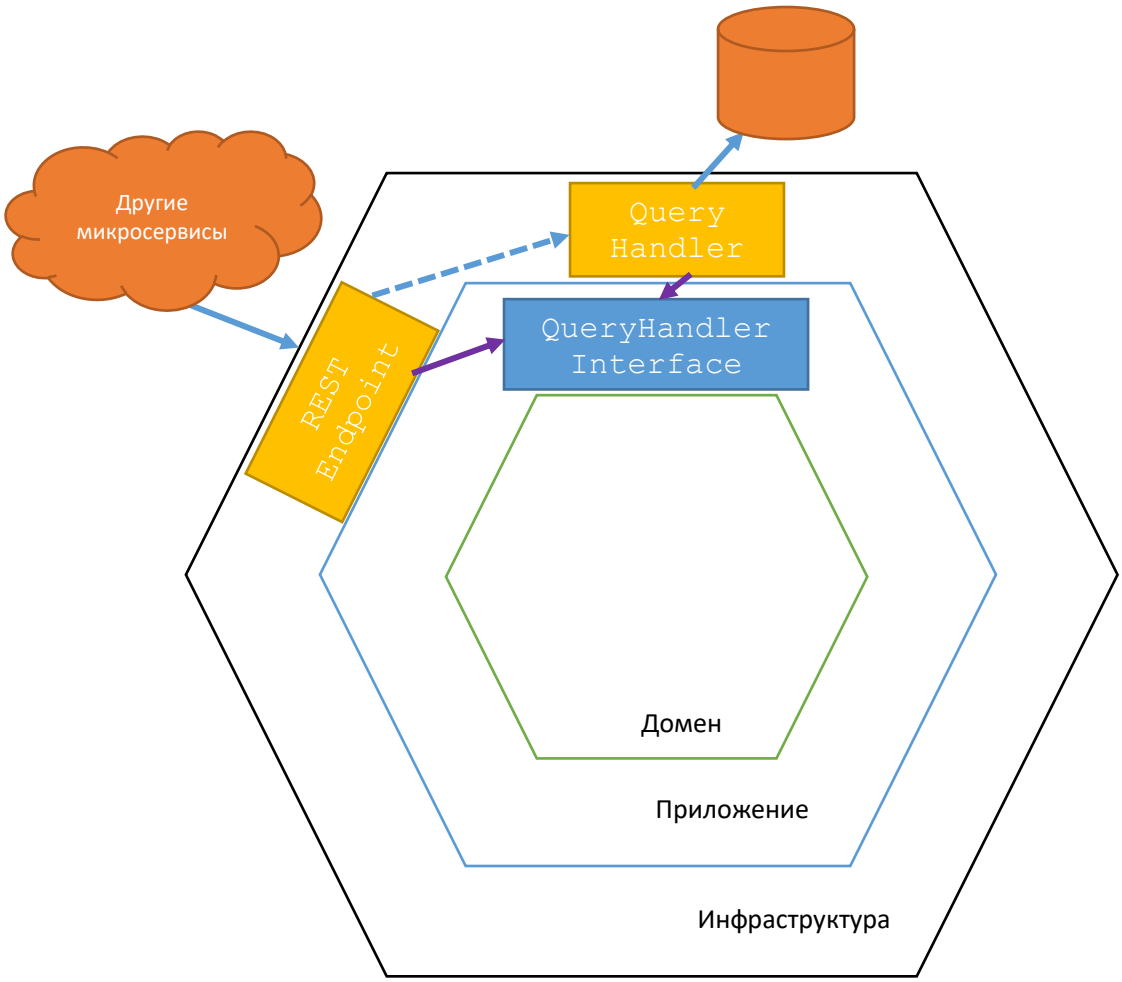

Рисунок 3 – Прохождение запроса

В отличие от обработчика команд, который с уровня приложения дальше обращается к домену, а сам инкапсулирует логику управления транзакциями, обработчик запросов на уровне приложения представлен интерфейсом, который может быть реализован на инфраструктурном уровне, например, прямыми запросами к БД.

Ключевой смысл использования интерфейса в данном случае состоит не совсем в инверсии зависимостей. В конце концов, запрос пришёл на инфраструктурный уровень, и обработчик на нём же, можно было бы обойтись и прямым вызовом. Однако, тогда это жёстко свяжет точку входа с реализацией обработчика запроса.

В случае же применения интерфейса уровня приложения реализация может быть безболезненно изменена, причем на реализацию любого уровня, даже на дальнейшее движение вглубь домена. Ну и про тестирование тоже не стоит забывать – с интерфейсом использование фейковой реализации тривиально.

Подробнее про CQRS, например, в [3, стр. 264] или [4].

Список использованных источников

1. Фаулер, М. Шаблоны корпоративных приложений. : Пер. с англ. – М. : ООО «И.Д. Вильямс», 2016. – 544 с. : ил. – Парал. тит. англ.
2. Buenosvinos, C. et al. Domain-Driven Design in PHP. – Packt, 2017. – 387 p.
3. Ричардсон, К. Микросервисы. Паттерны разработки и рефакторинга. – СПб.: Питер, 2019. – 544 с.: ил. – (Серия «Библиотека программиста»).
4. CQRS, Event Sourcing, and Domain-Driven Design FAQ [Электронный ресурс]. – Режим доступа: https://cqrs.nu/. – Дата доступа: 23.02.2020.