Система TDG:  
руководство по эксплуатации

## Оглавление

[1. Введение 4](#_Toc16248672)

[1.1. Общие сведения о системе и руководстве 4](#_Toc16248673)

[1.2. Требования к персоналу 4](#_Toc16248674)

[2. Установка и запуск приложения 6](#_Toc16248675)

[2.1. Предварительная настройка серверов 6](#_Toc16248676)

[2.2. Установка 6](#_Toc16248677)

[2.3. Директории по умолчанию 9](#_Toc16248678)

[3. Настройка кластера 10](#_Toc16248679)

[3.1. Создание набора реплик и задание ролей 10](#_Toc16248680)

[3.2. Запуск кластера 16](#_Toc16248681)

[3.3. Загрузка конфигурации системы 16](#_Toc16248682)

[3.4. Автоматическая синхронизация настроек в кластере 16](#_Toc16248683)

[3.5. Токен кластера 17](#_Toc16248684)

[4. Роли 17](#_Toc16248685)

[4.1. Рекомендации по назначению ролей на инстансах 19](#_Toc16248686)

[5. Основной процесс обработки запроса 20](#_Toc16248687)

[6. Ремонтная очередь 22](#_Toc16248688)

[6.1. Нотификации 24](#_Toc16248689)

[7. Управление бизнес-объектами 25](#_Toc16248690)

[7.1. Expiration 25](#_Toc16248691)

[7.2. Delete Aggregate 25](#_Toc16248692)

[8. Логирование 27](#_Toc16248693)

[9. Репликация объектов 29](#_Toc16248694)

[10. Задачи 31](#_Toc16248695)

[11. Конфигурация системы (config.yml) 33](#_Toc16248696)

[11.1. Секции 36](#_Toc16248697)

[12. Разработка приложений 43](#_Toc16248698)

[12.1. Разработка доменной модели 43](#_Toc16248699)

[12.2. Программный интерфейс репозитория 56](#_Toc16248700)

[12.3. Представление модели 60](#_Toc16248701)

[12.4. Разработка тестов 62](#_Toc16248702)

[13. Доступ к данным 66](#_Toc16248703)

[13.1. Язык запросов 66](#_Toc16248704)

[14. Приложение 71](#_Toc16248705)

[14.1. Скрипт tdgctl.py 71](#_Toc16248706)

## 1. Введение

### 1.1. Общие сведения о системе и руководстве

Система TDG (Tarantool Data Grid) предназначена для хранения сложных бизнес-объектов и выполнения аналитических запросов над ними в реальном времени.

Настоящее руководство содержит описание функциональных характеристик системы TDG и информацию, необходимую для ее установки и эксплуатации.

### 1.2. Требования к персоналу

Руководство ориентировано на пользователей, владеющих основными навыками:

* Apache Avro — двоичный формат представления данных. Получил своё развитие как подпродукт проекта Hadoop. Avro применяет схему для задания структуры кодируемых данных. Язык описания схемы представляет собой json-файл специальной структуры. Эта система сериализации данных является самой эффективной по плотности сжатия данных, однако она обязательно требует наличия схемы, как для сериализации данных, так и для их десериализации.

Пример схемы:

* {
* "type": "record",
* "name": "Person",
* "fields": [
* {"name": "userName", "type": "string"},
* {"name": "favoriteNumber", "type": ["null",
* "long"], "default": null},
* {"name": "interests", "type": {"type":
* "array", "items": "string"}}
* ]
* }
* JSON - альтернатива XML, которую использует javascript
* GraphQL - язык запросов, который описывает как получить данные с сервера, и используется в API для загрузки данных с сервера на клиент
* Linux – операционная система
* Принципы конструирования ИС
* Работа распределенных систем (работа с кластером)

## 2. Установка и запуск приложения

Процедура установки и запуска приложения в кластере включает в себя следующие этапы:

* [Предварительная настройка](#initialsetup)
* [Установка](#ustanovka)
* [Настройка кластера](#cluster-setup)

### 2.1. Предварительная настройка серверов

Система TDG поддерживает операционные системы **Red Hat Enterprise Linux** и **CentOS** версий 7.5 и выше.

* + - Примечание
    - Приложение может быть запущено на других Linux-дистрибутивах, основанных на systemd, но не тестируется на них и может не соответствовать заявленной функциональности.

1. Пользователь, под которым будет происходить установка, должен иметь привилегии root. Также нужно отключить для него запрос пароля в файле /etc/sudoers:

* admin ALL=(ALL:ALL) NOPASSWD: ALL

1. Необходимо включить ssh-сервер.
2. Должен быть загружен публичный ключ администратора для доступа на машину по ssh-протоколу.
3. Для работы скрипта установки требуется Python 3 и библиотека fabric:

* pip3 install fabric

### 2.2. Установка

#### 2.2.1. Дистрибутив установки

Загрузите актуальный дистрибутив для установки приложения.

Нам потребуется образ установки в формате tar-архива: tdg-<version>.tar.gz, где <version> — версия релиза.

#### 2.2.2. Подготовка файла конфигурации кластера

Для установки приложения на серверы также требуется подготовить файл конфигурации кластера в формате JSON.

Рассмотрим структуру и параметры файла на примере:

{  
 "general":  
 {  
 "cluster\_cookie": "some\_secret\_cookie"  
 },  
 "servers":  
 [  
 {  
 "address": "172.19.0.2",  
 "username": "vagrant",  
 "instances":  
 [  
 {  
 "name": "input\_processor",  
 "binary\_port": 3000,  
 "http\_port": 8080,  
 "memory\_mb": 128  
 },  
 {  
 "name": "logger\_etc",  
 "binary\_port": 3001,  
 "http\_port": 8081,  
 "memory\_mb": 128  
 },  
 {  
 "name": "storage\_1",  
 "binary\_port": 3002,  
 "http\_port": 8082,  
 "memory\_mb": 1024  
 },  
 {  
 "name": "storage\_2\_replica",  
 "binary\_port": 3003,  
 "http\_port": 8083,  
 "memory\_mb": 1024  
 }  
 ]  
 },  
 {  
 "address": "172.19.0.3",  
 "username": "vagrant",  
 "instances":  
 [  
 {  
 "name": "storage\_2",  
 "binary\_port": 3004,  
 "http\_port": 8084,  
 "memory\_mb": 1024  
 },  
 {  
 "name": "storage\_1\_replica",  
 "binary\_port": 3005,  
 "http\_port": 8085,  
 "memory\_mb": 1024  
 }  
 ]  
 }  
 ]  
}

Здесь указан массив серверов servers, на которые будет происходить установка. Для каждого сервера нужно указать:

* username — имя пользователя, от которого будет происходить установка;
* address — адрес сервера;
* instances — список устанавливаемых инстансов.

Для каждого инстанса задается имя name. При установке на физические Linux-сервера это будет имя юнита systemd. Для каждого инстанса указана пара портов — binary\_port и http\_port, которые должны быть открыты на сервере для взаимодействия инстансов. http\_port требует протокола TCP, а binary\_port — как TCP, так и UDP.

Параметр memory\_mb ограничивает потребление памяти Tarantool внутри инстанса и задается в мегабайтах.

В параметре cluster\_cookie задается [токен кластера](#cluster-cookie), который необходим для того, чтобы члены кластера могли общаться между собой и при этом не допустить попадание в кластер постороннего члена.

Все перечисленные параметры обязательны.

#### 2.2.3. Запуск скрипта установки

Для установки системы используется скрипт tdgctl.py. См. подробнее о [формате команд скрипта](#tdgctl).

При запуске скрипта нужно указать команду deploy и передать аргументами путь к файлу с конфигурацией кластера и файлу с образом установки:

./tdgctl.py -c deploy.json deploy <image\_name>.tar.gz

где deploy.json — файл с конфигурацией кластера, <image\_name>.tar.gz — файл с образом установки.

После этого скрипт подключится к серверам по списку, загрузит образ и запустит инстансы.

Скрипт tdgctl.py также позволяет выполнять другие операции в кластере, такие как остановка и старт инстансов, удаление инстансов, сбор логов и др.

### 2.3. Директории по умолчанию

Рабочие директории {workdir} по умолчанию, в которых сохраняются данные для каждого инстанса: /var/lib/tarantool/<instance\_name>, где <instance\_name> — имя инстанса, определенное в [файле конфигурации кластера](#cluster-config-code) (в нашем примере — «input\_processor», «storage\_1» и т.д.).

## 3. Настройка кластера

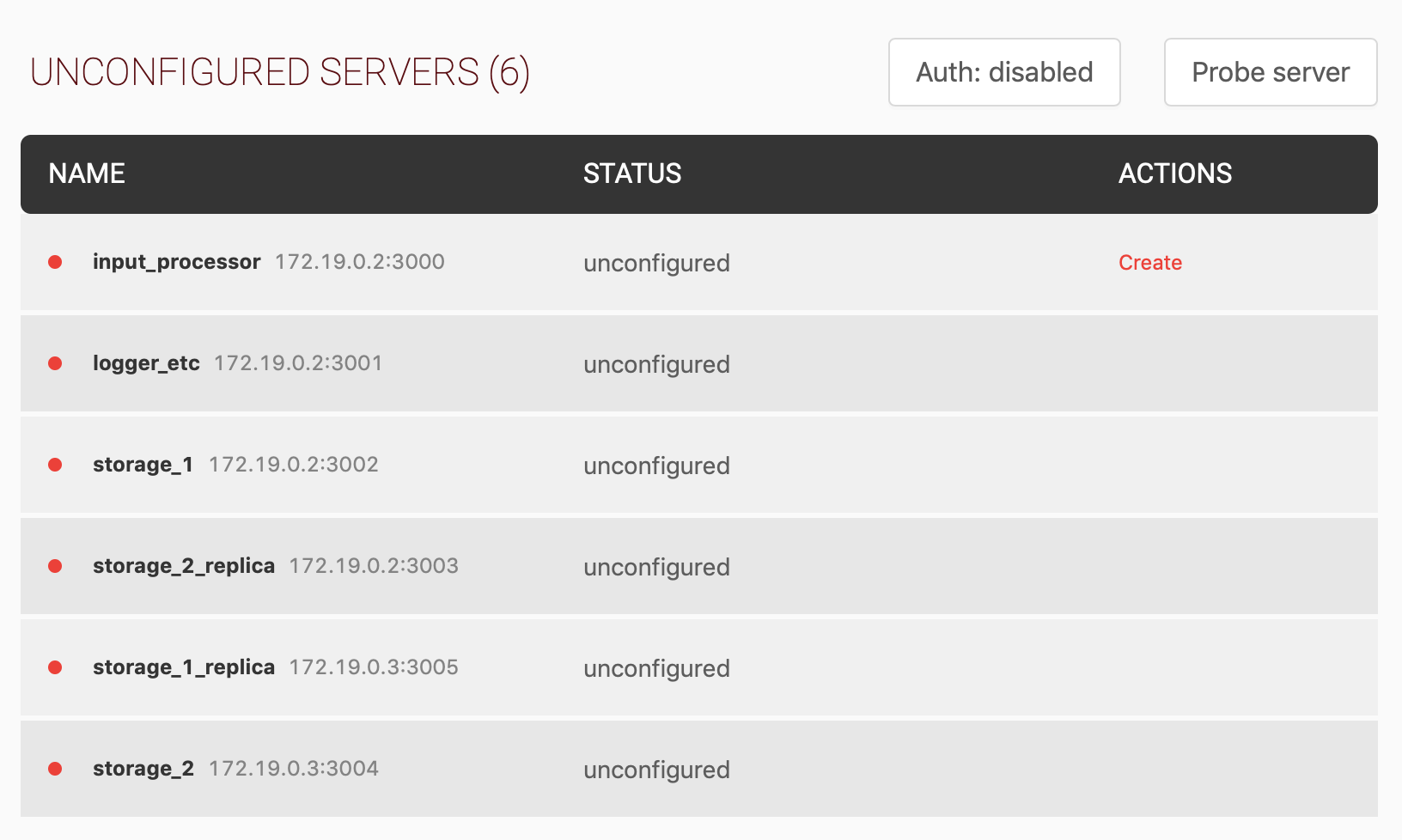
После успешной установки приложения на серверы необходимо настроить кластер, что включает в себя следующие действия:

* [Создание набора реплик (репликасетов) и задание ролей инстансов](#repicasetscreation)
* [Запуск кластера](#vshard-bootstrap)
* [Загрузка конфигурации системы](#system-config-upload)

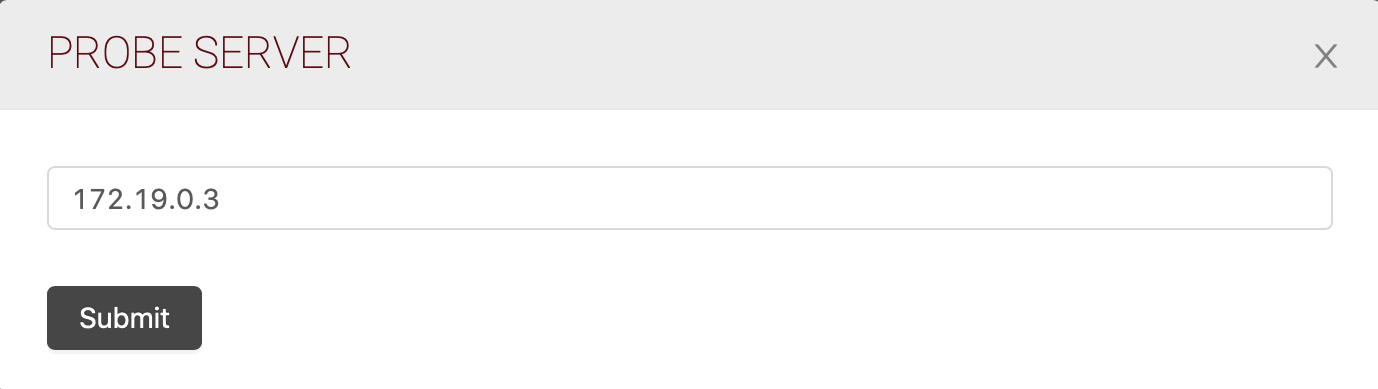
### 3.1. Создание набора реплик и задание ролей

Операции по настройке кластера можно выполнить через web-интерфейс администратора. Для подключения к web-интерфейсу надо зайти в браузере на любой сервер по доступному на нем http-порту http://<instance\_address>:<instance\_http\_port> (указаны в [файле конфигурации кластера](#cluster-config-code)). В рассматриваемом примере это может быть http://172.19.0.2:8080 (соответствует инстансу с именем «input\_processor»).

На вкладке **Cluster** мы имеем набор не сконфигурированных инстансов, на что указывает их текущий статус «unconfigured»:

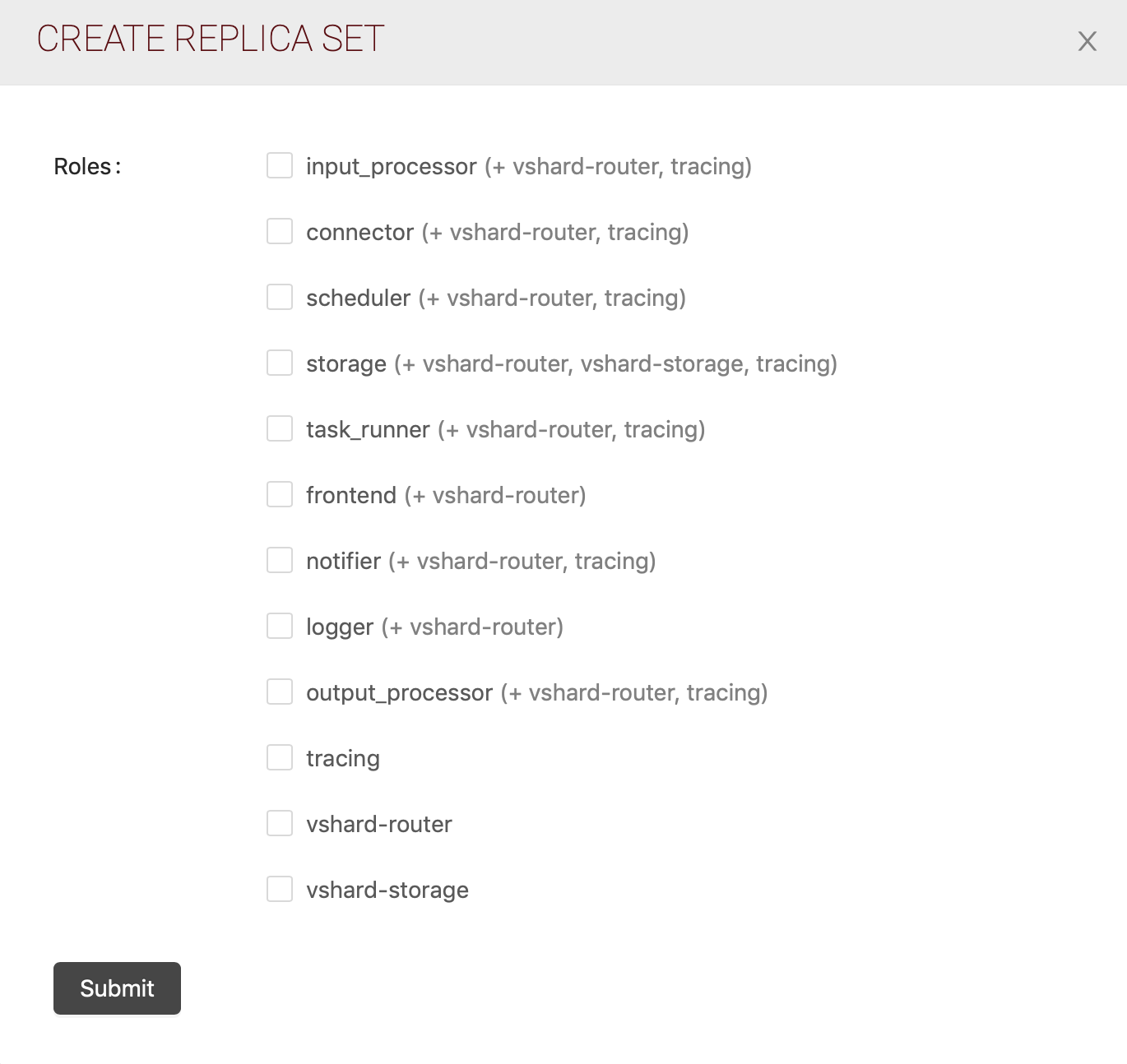


* + - Примечание
    - В некоторых подсетях может не работать автоматический поиск доступных серверов (autodiscovery). Тогда инстансы этих серверов могут отсутствовать в таблице. Для ручной проверки доступности можно использовать функцию **Probe server** (кнопка справа над таблицей с инстансами). Функция проверяет, доступен ли сервер, и, если доступен, инстансы этого сервера появятся в таблице.



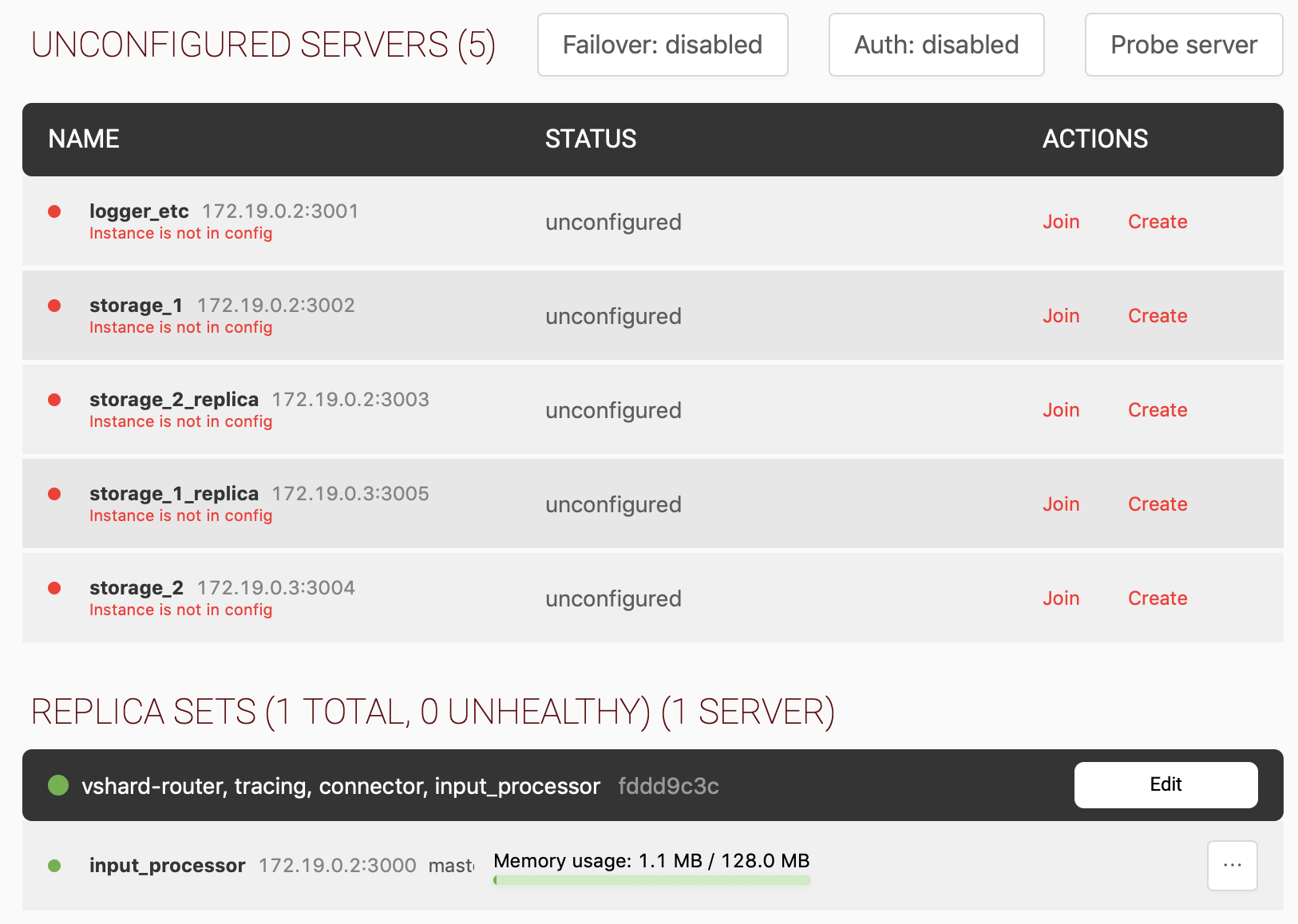
Далее необходимо создать наборы реплик (репликасеты) и задать роли инстансов.

Для инстанса, через который мы зашли в web-интерфейс (в нашем примере — «core»), в колонке **ACTION** доступна кнопка **Create** для создания репликасета и выбора ролей. По нажатию этой кнопки появится диалог создания репликасета:



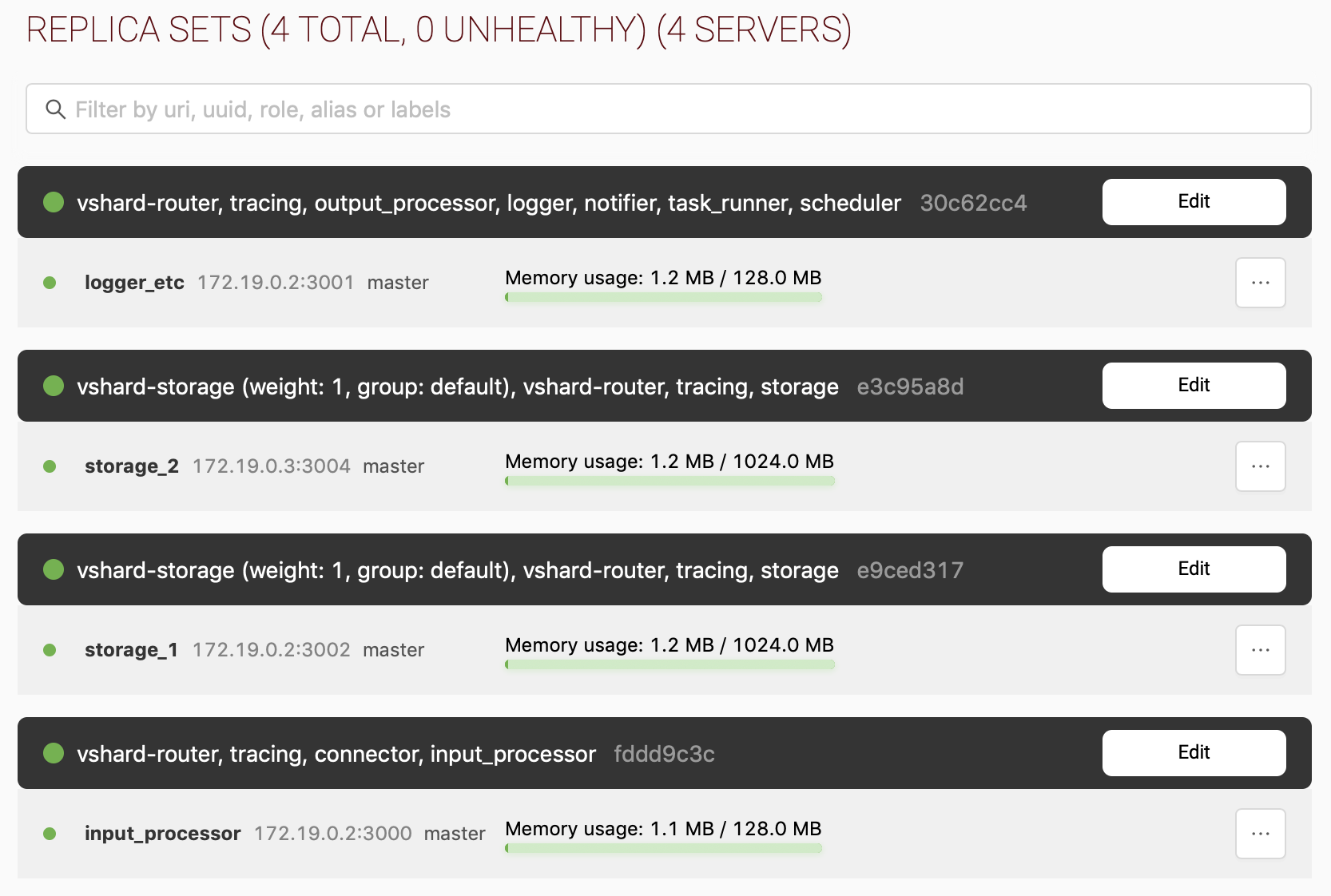
Выберите необходимую роль(и) и нажмите **Submit**. Подробнее о ролях и рекомендациях по их назначению см. раздел [«Роли»](#roles).

После определения роли инстанс становится частью кластера в составе созданного репликасета (таблица **REPLICA SETS**):



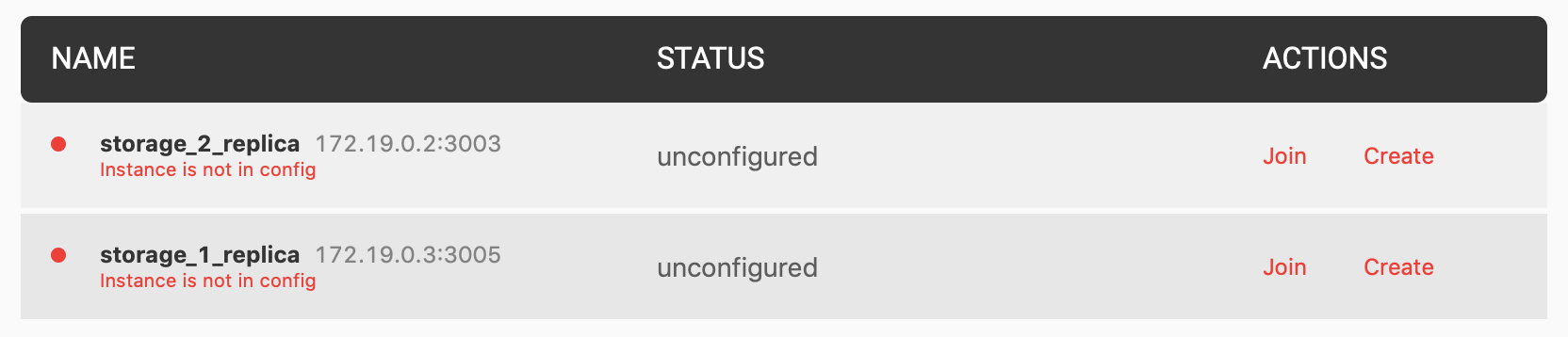
Далее необходимо настроить оставшиеся не сконфигурированные инстансы — либо создав новые реликасеты (кнопка **Create**), либо включив их в состав уже имеющихся (кнопка **Join**).

В нашем примере мы создадим 3 дополнительных репликасета:

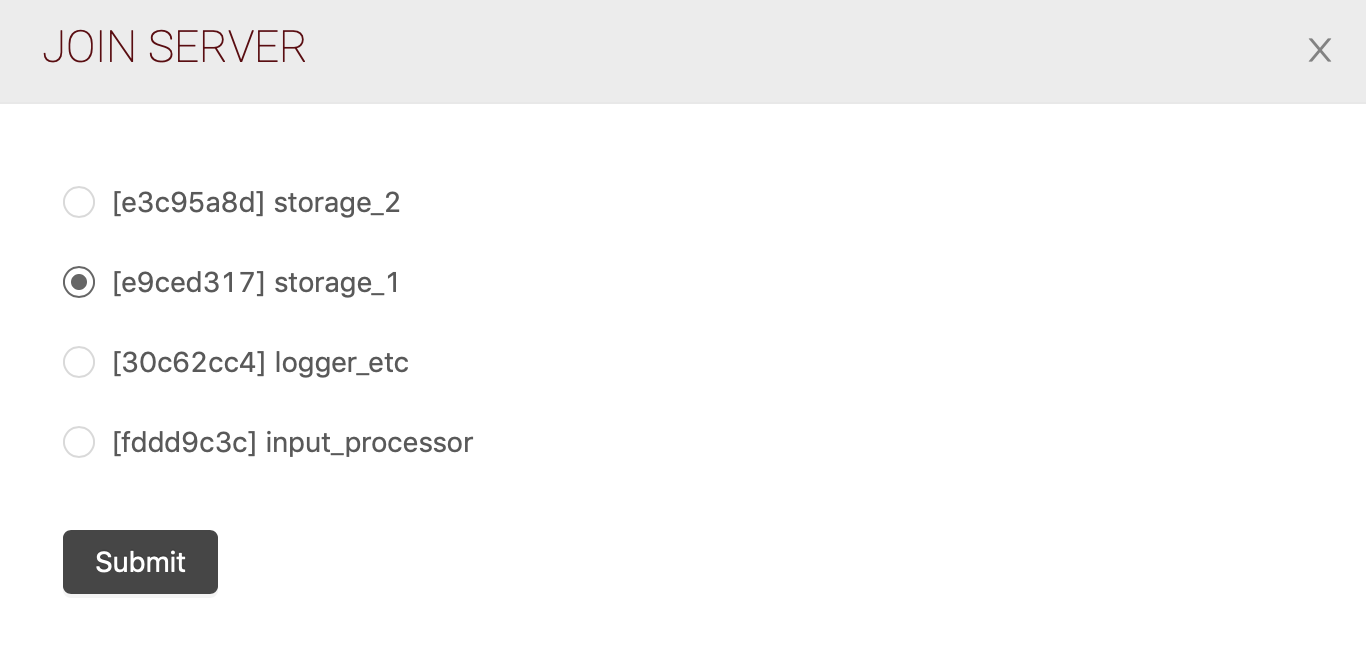


* + - Примечание
    - Когда будет задан первый инстанс с ролью storage, в web-интерфейсе появится секция для инициализации модуля «Tarantool vshard», который обеспечивает механизм горизонтального масштабирования. Его нужно инициализировать позже, после того, как будут настроены все репликасеты.

В нашем примере конфигурации осталось два ненастроенных инстанса, находящихся на разных серверах:

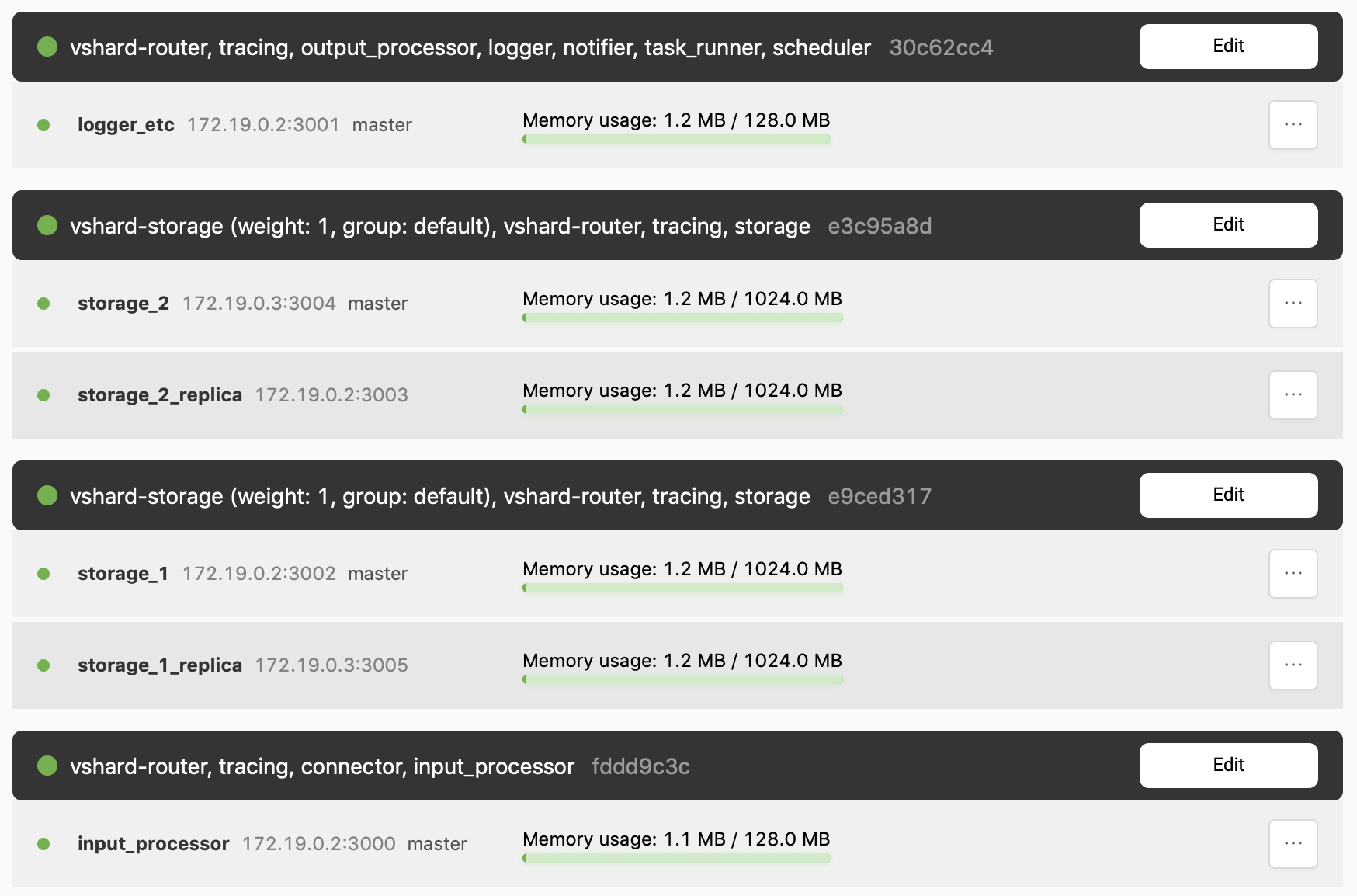


Мы включим их в уже созданные репликасеты роли storage, определив как реплики мастеров. Для этого нажмите кнопку **Join** для одного из инстансов и в диалоге **JOIN SERVER** выберите нужный репликасет, в который собираетесь добавить инстанс. Нажмите **Submit**.



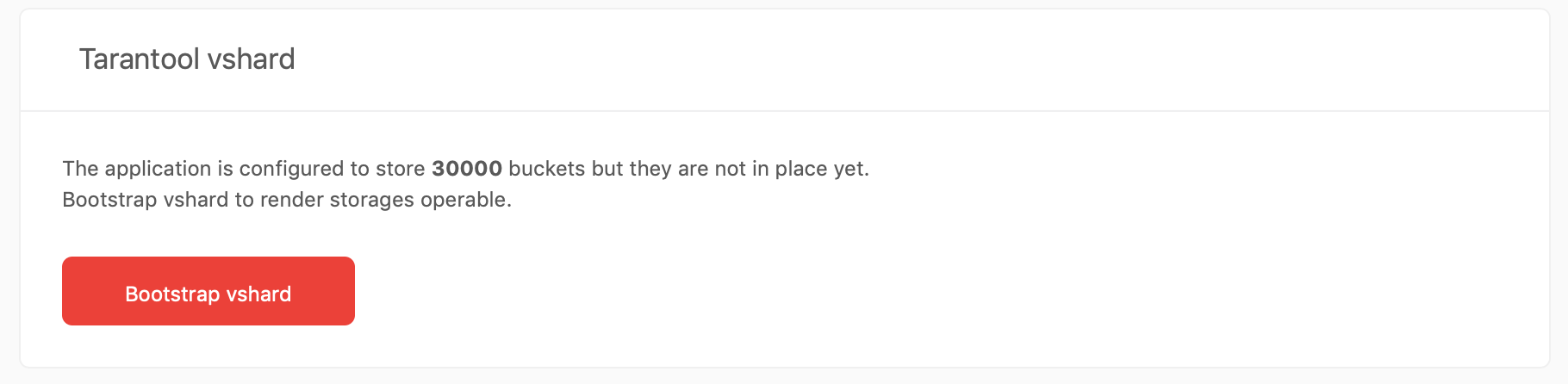
Инстанс будет включен в состав репликасета в качестве реплики. Выполните аналогичную операцию для второго инстанса.

Ниже показан итог настройки кластера для нашего примера после всех вышеупомянутых операций:



### 3.2. Запуск кластера

После создания репликасетов и задания ролей инстансов нужно запустить кластер в работу, выполнив инициализацию модуля «Tarantool vshard». Пока кластер не запущен, на вкладке **Cluster** отображается секция **Tarantool vshard**.



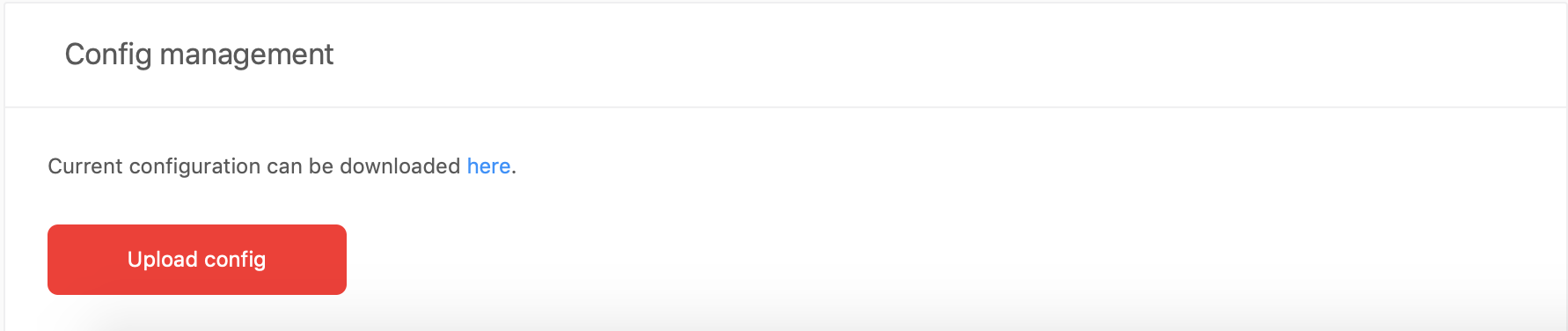
Для инициализации нажмите кнопку **Bootstrap vshard**. При этом будут созданы виртуальные сегменты (virtual buckets) и распределены по хранилищам с учетом количества инстансов с ролью storage. Подробнее про принцип работы модуля см. в [документации по Tarantool](https://www.tarantool.io/ru/doc/2.1/reference/reference_rock/vshard/)

### 3.3. Загрузка конфигурации системы

Последний этап запуска системы в работу — загрузка и применение ее конфигурации.

Для этого через web-интерфейс (**Cluster > Config management > Upload config**) необходимо загрузить архив .zip, который должен содержать основной файл конфигурации системы (см. руководство по техническому сопровождению и поддержке) config.yml и другие необходимые файлы (описание модели данных, Lua-код функций и др.).

В дальнейшем актуальную конфигурацию (config.yml и сопутствующие файлы) всегда можно скачать в этой же секции **Config management** по ссылке «Current configuration can be downloaded here».



После успешной загрузки конфигурации и ее применения система готова к работе.

### 3.4. Автоматическая синхронизация настроек в кластере

Для нормального функционирования кластера все инстансы должны иметь одинаковые настройки. С этой целью каждый инстанс хранит копию конфигурации config.yml в своей рабочей директории /var/lib/tarantool/<instance\_name>. Также в этой директории хранятся резервная копия config.backup.yml и [токен кластера](#cluster-cookie) .tarantool.cookie. Кластер синхронизирует эти файлы настроек у всех инстансов.

После первоначальной загрузки файл конфигурации config.yml автоматически обогащается информацией о топологии кластера с указанием всех серверов, репликасетов, инстансов и их ролей. Также добавляется информация о созданных спейсах для хранения объектов согласно модели данных. Именно в таком виде конфигурация хранится в рабочей директории каждого инстанса и синхронизируется между ними.

При дальнейшей настройке системы — добавлении пользователей и токенов приложений, настройке прав доступа и др. — информация об этом добавляется в соответствующие секции файла конфигурации и синхронизируется в кластере. Аналогичная синхронизация выполняется при любых изменениях текущих настроек.

### 3.5. Токен кластера

Токен кластера — это некий уникальный идентификатор, одинаковый для всех нод кластера и хранящийся на каждой из нод. Он шифрует UDP-трафик построения кластера и необходим для того, чтобы все члены кластера могли общаться между собой и не допускать в кластер посторонних членов. Если у какой-либо ноды токен другой, остальные ноды кластера ее не видят, и сама нода не видит кластер.

Токен кластера можно задать, например, при установке системы, определив в [файле конфигурации кластера](#cluster-config-code) параметр cluster\_cookie. Если токен кластера не задан, то берется значение по умолчанию.

Токен кластера хранится в файле .tarantool.cookie.

Поиск файла .tarantool.cookie осуществляется в следующих местах по убыванию важности:

* {workdir}/.tarantool.cookie, где {workdir} задается или в параметре инициализации, или из переменной среды $WORKDIR, или берется текущая директория.
* $HOME/.tarantool.cookie, где $HOME — переменная среды.

## 4. Роли

В кластере функции инстансов распределяются на основе ролей. Кластерные роли — это Lua-модули, которые реализуют специфическую для инстанса логику.

Существуют 2 вида ролей: **встроенные** и **пользовательские** (настраиваемые). Встроенные роли, vshard-router и vshard-storage, и логика их работы уже [определены в Tarantool Enterprise](https://www.tarantool.io/en/enterprise_doc/1.10/dev/#built-in-roles), на базе которого построена система TDG, и не требуют дополнительной конфигурации. Логику работы пользовательских ролей необходимо дополнительно [сконфигурировать](#_11._Конфигурация_системы). Это выполняется администратором.

В системе TDG реализованы следующие пользовательские роли:

* connector – прием запросов по сети на обработку;
* input\_processor – обработка запроса;
* storage – хранение данных (обработанных объектов);
* logger – сбор логов (со всех инстансов всех ролей);
* notifier – отправка нотификаций о событиях в ремонтной очереди;
* output\_processor – преобразование объекта в формат сторонней системы и отправка объекта в эту систему;
* task-runner – выполнение задач;
* scheduler – старт задач (по расписанию или вручную) на инстансе с ролью task-runner;
* frontend – служит для загрузки статического расширения пользовательского интерфейса.

Каждая из пользовательских ролей требуют также подключения встроенных ролей. Встроенные роли не нужно назначать инстансам специально, они подключаются автоматически к пользовательским ролям при назначении последних — см. таблицу ниже. Также вместе с большинством пользовательских ролей автоматически подключается дополнительная роль tracing, которая нужна для мониторинга производительности модулей системы.

|  |  |
| --- | --- |
| Пользовательская роль | Автоматически подключаемые роли |
| connector | vshard-router, tracing |
| input\_processor | vshard-router, tracing |
| storage | vshard-router, vshard-storage, tracing |
| logger | vshard-router |
| notifier | vshard-router, tracing |
| output\_processor | vshard-router, tracing |
| scheduler | vshard-router, tracing |
| task\_runner | vshard-router, tracing |
| frontend | vshard-router |

### 4.1. Рекомендации по назначению ролей на инстансах

Хотя в системе нет понятия «обязательная роль», имеет смысл говорить о минимально необходимом наборе ролей — connector, input\_processor, storage, который обеспечивает основные функции системы — получение объекта, его обработку и хранение в системе. Остальные роли назначаются по необходимости, в зависимости от решаемых бизнес-задач.

Роли имеют ряд характеристик, в зависимости от которых их можно делить на различные группы:

* по концепции хранения состояния: не хранят состояние (stateless) / хранят состояние (stateful);
* по количеству экземпляров в кластере: singleton / произвольное количество экземпляров.

|  |  |  |
| --- | --- | --- |
| Роль | Stateless / Stateful | Singleton |
| connector | stateless | нет |
| input\_processor | stateless | нет |
| storage | stateful | нет |
| logger | stateful | да |
| notifier | stateful | да |
| output\_processor | stateless | нет |
| scheduler | stateful | да |
| task\_runner | stateless | нет |

Все роли являются реплицируемыми (репликасет с ролью может содержать более одной ноды).

На одном инстансе можно назначить одну или несколько ролей.

Учитывая вышесказанное, можно дать следующие рекомендации по организации кластера и назначению ролей на инстансах кластера:

1. Роли connector, input\_processor, task\_runner и output\_processor задействованы в обработке объектов. Их нужно горизонтально масштабировать пропорционально входящей нагрузке и утилизации CPU на серверах.
2. Если вы создаете на инстансе роль connector или input\_processor, обычно рекомендуется создать на этом же инстансе и вторую роль из этой пары. Исключение может быть в случае, когда первоначальная обработка объекта на роли connector является трудозатратной операцией и имеет смысл масштабировать эту роль отдельно.
3. Роль storage (хранит состояние, не singleton): создание репликесета с нодами в разных дата-центрах. Количество репликасетов (горизонтальное масштабирование) — пропорционально объему данных, хранимых в RAM.
   * + Примечание
     + Если инстансу назначена роль storage, то после этого исключить (expel) из кластера этот инстанс будет нельзя.
4. Роли logger, notifier, scheduler (хранят состояние, singleton): обычно достаточно репликасета с 2 нодами, разнесенными по разным дата-центрам.
5. Роль output\_processor не хранит состояние напрямую. Функционально роль реплицирует объекты во внешние системы, но для хранения объектов, предназначенных для репликации, а также объектов, чья репликация завершилась с ошибкой, используется роль storage. Поэтому логика горизонтального масштабирования роли output\_processor аналогична логике для ролей, используемых для обработки объектов (см. п.1 выше).

## 5. Основной процесс обработки запроса

Роль connector принимает запрос (объект) из системы-источника. Исходный формат входящего объекта — JSON или XML, в зависимости от системы-источника. Если инстансов с ролью connector несколько, выбор инстанса и балансировка нагрузки выполняется сервисом nginx.

На роли connector происходит первоначальная обработка (parsing) входящего объекта: объект преобразуется в Lua-объект (Lua-таблицу) и направляется на роль input\_processor. Также для каждого запроса генерируется его UUID, по которому в дальнейшем можно проследить весь путь объекта.

Дальнейший маршрут объекта определяется в соответствие с ключом маршрутизации (routing\_key), который присваивается объекту на разных этапах его обработки. Логика маршрутизации по ключу и порядок обработки объекта задается в файле конфигурации системы config.yml.

На роли input\_processor объект проходить через конвейеры обработки (pipelines, пайплайны), которые определены в конфигурации config.yml.

1. Прежде всего выполняется классификация объекта. Если классификация успешна, объекту присваивается ключ маршрутизации, определяющий дальнейшие пайплайны для обработки.
2. Если задано в конфигурации, объект проходит обработку в других пайплайнах в соответствие с ключом маршрутизации.
3. Объект валидируется в соответствие с моделью данных.

Если на любом из этапов — a, b, c — происходит ошибка, объект остается в ремонтную очередь. Информация об объектах в [ремонтной очереди](#repair-queue) с описанием ошибок доступна администратору через web-интерфейс. Если определена роль notifier и настроена конфигурация mail server и subscribers, подписчикам будет отправлена нотификация об ошибке.

Если классификация, обработка и валидация объекта прошли успешно, объекту присваивается соответствующий routing\_key, и объект сохраняется на инстансе с ролью storage. Тип сохраняемого объекта также определяется настройками в config.yml в соответствие с ключом маршрутизации объекта.

Дальнейшие возможные действия с сохраненными объектами:

* запрашиваются внешними системами;
* [реплицируются](#output_processor) во внешние системы;
* действия с объектами в рамках [задач](#tasks);
* [удаление](#aggregate-mgnt) объектов.

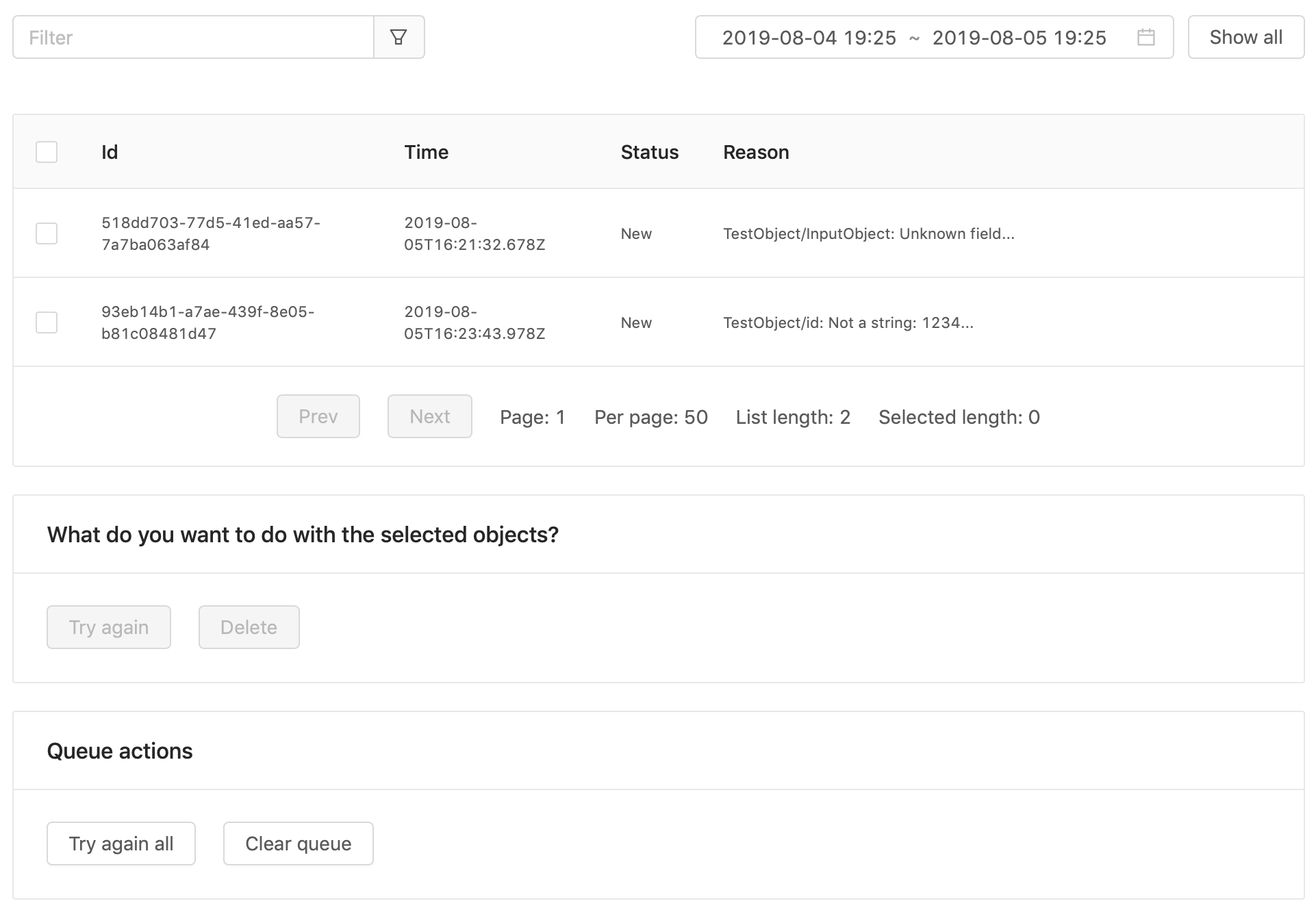
## 6. Ремонтная очередь

Когда объект поступает в систему на обработку, он сразу помещается в ремонтную очередь. Если объект удалось обработать и сохранить, он удаляется из ремонтной очереди. В случае ошибки объекты остаются в ремонтной очереди, и администратор имеет возможность просматривать их и после устранения источника проблемы отправлять на повторную обработку.

Можно выделить следующие основные причины возникновения ошибок, когда объекты остаются в ремонтной очереди:

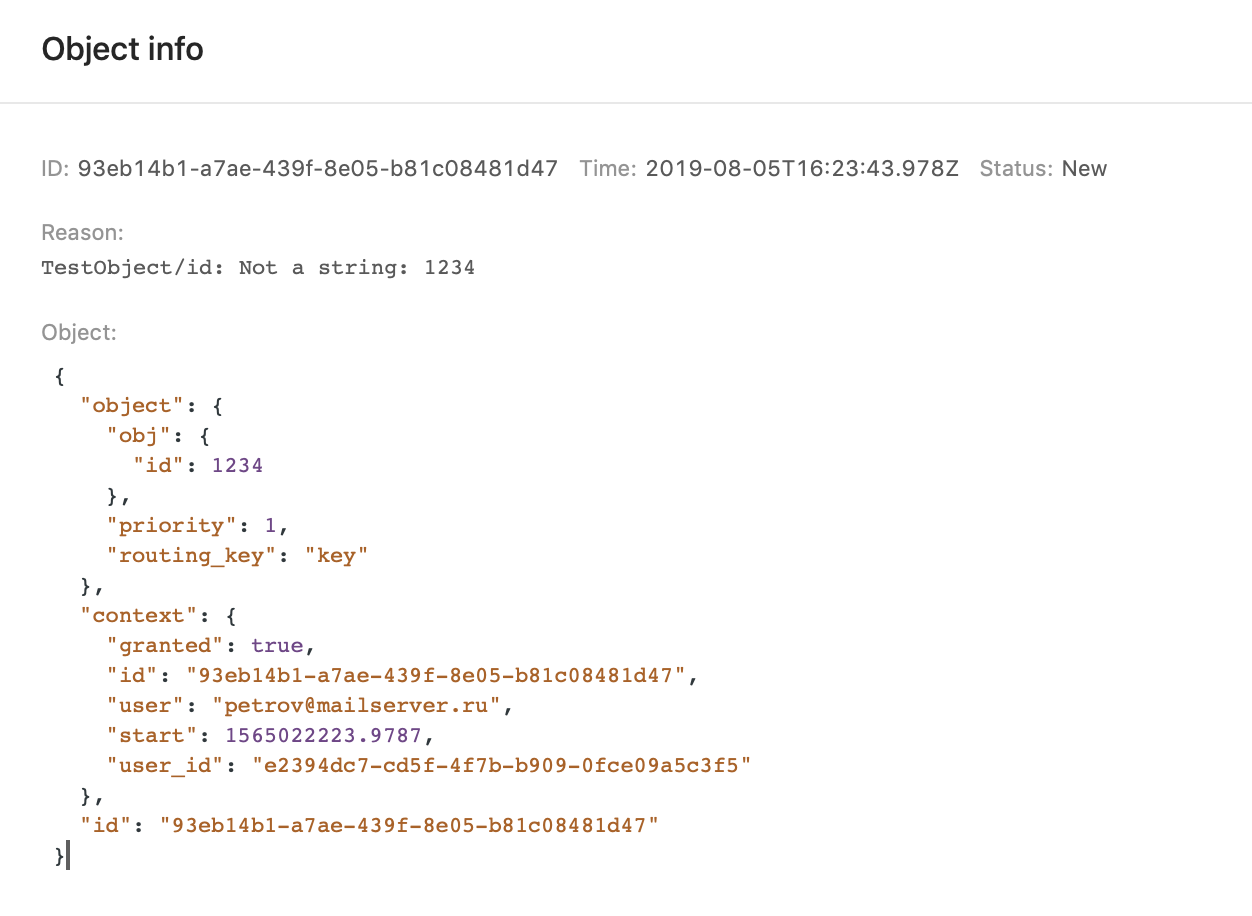
* Ошибка при обработке объекта в каком-либо из пайплайнов.
* Система TDG ожидает объект в определенном формате, но объект пришел из внешней системы в другом формате.
* Внутренняя ошибка системы.
* Сбой оборудования (hardware).

Работать с объектами в ремонтной очереди можно через web-интерфейс на вкладке **Repair**.



В таблице отображается текущий список объектов в ремонтной очереди. Двойной клик на объект в таблице открывает отдельное окно **Object info** со следующей информацией об объекте:

* Id — UUID объекта;
* Time — дата и время, когда объект был помещен в ремонтную очередь;
* Status — статус объекта в ремонтной очереди (возможные значения: «New», «In Progress», «Reworked»);
* Reason — описание причины ошибки и полный stack trace;
* Object — текущая структура объекта в формате JSON.



Для поиска нужного объекта есть возможность фильтрации по любому сочетанию символов в любой колонке таблицы — поле Filter; по дате и времени — поле Start Time ~ End Time.

Доступные действия над объектами в ремонтной очереди:

* **Try again** — повторная обработка объекта той же функцией пайплайна, при выполнении которой возникла ошибка;
* **Delete** — удаление объекта из ремонтной очереди.

Когда объект попадает в ремонтную очередь, он имеет статус «New». При повторной обработке статус объекта меняется на «In Progress». Если обработка прошла успешна, объект удаляется из ремонтной очереди. Если при повторной обработке опять возникла ошибка, система выдаст сообщение об ошибке, и объект останется в ремонтной очереди со статусом «Reworked».

Аналогично действиям над отдельными объектами, можно выполнить действия над всеми объектами в ремонтной очереди:

* **Try again all** — повторная обработка всех объектов;
* **Clear queue** — удаление всех объектов из ремонтной очереди.

### 6.1. Нотификации

В системе есть возможность информировать пользователей о попадании объекта в ремонтную очередь. Для этого должна быть определена роль notifier, а также заданы настройки почтового сервера и подписчиков, которым будут отправляться нотификации.

Роль notifier задается на одном из инстансов при [настройке ролей в кластере](#repicasetscreation). Настройки почтового сервера и подписчиков задаются через web-интерфейс на вкладках **Settings > Mail** server и **Settings > Subscribers** соответственно.

#### 6.1.1. Settings > Mail server

Настройки:

* Url — сервер SMTP, используемый для отправки нотификаций;
* From — отправитель, который будет показан в почтовом клиенте;
* User name — имя пользователя сервера SMTP;
* Password — пароль пользователя сервера SMTP;
* Timeout (sec) — тайм-аут запроса к серверу SMTP, в секундах.

#### 6.1.2. Settings > Subscribers

Необходимо создать подписчиков (кнопка **Create subscriber**), которые будут получать нотификации, указав их имя и Email. Возможные действия с подписчиками аналогичны действиям с пользователями. Можно

* создать новых подписчиков;
* редактировать данные текущих подписчиков: изменить имя и Email;
* удалить подписчиков.

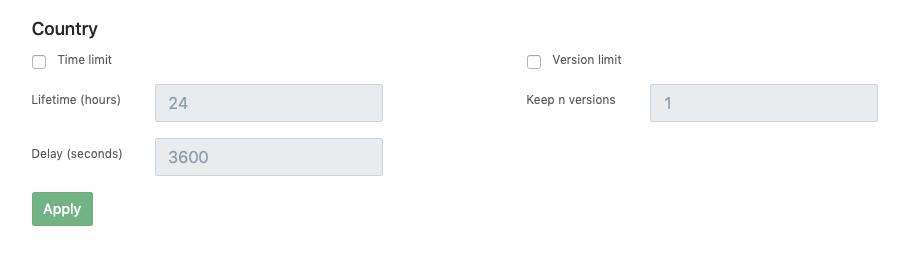
## 7. Управление бизнес-объектами

### 7.1. Expiration

В системе есть возможность сконфигурировать время жизни бизнес-объекта (агрегата). В конце жизни объект физически удаляется из системы. Также возможно удалять агрегаты вручную в любой нужный момент.

Эти возможности системы полезны при работы с типами объектов, которые нет необходимости хранить дольше определенного времени, например, суточные котировки и т.п.

Времени жизни объекта можно задать через web-интерфейс на вкладке **Expiration**. Объекты на этой вкладке описаны в модели данных и становятся доступны после загрузки модели в систему.



В секции **Time limit** задаются

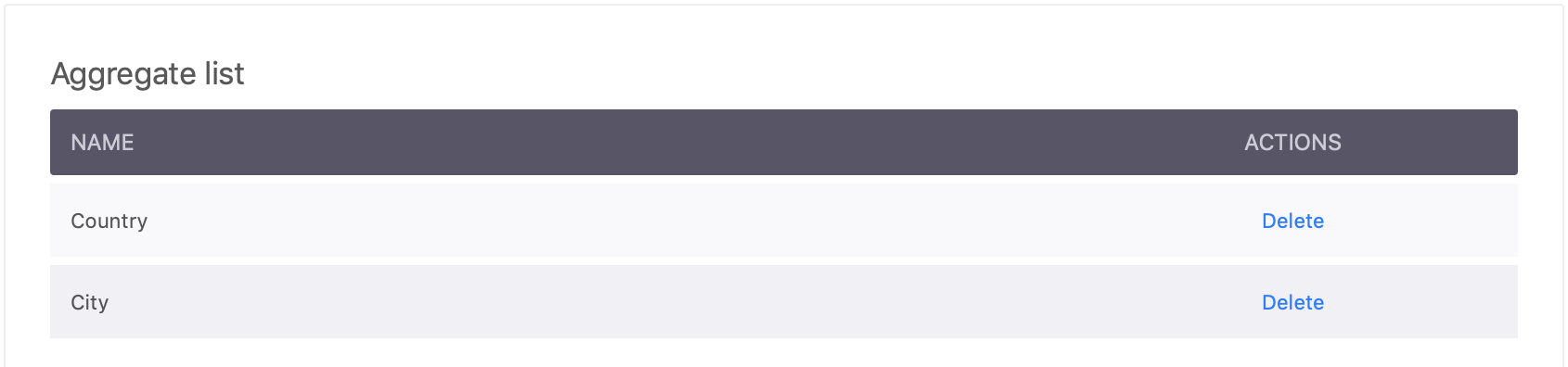
* Lifetime (hours) — время жизни объекта в часах. Значение по умолчанию: 24.
* Delay (seconds) — интервал в секундах, через который запускается очередная проверка устаревших объектов и их удаление. Значение по умолчанию: 36000.

В секции **Version limit**, параметр Keep n versions, также можно задать ограничение количества версий для объектов данного типа. Это ограничение срабатывает синхронно в момент вставки новой версии объекта. Значение по умолчанию: 1.

Ограничения времени жизни объекта и ограничение количества версий могут работать как по отдельности, так и одновременно.

### 7.2. Delete Aggregate

На вкладке **Settings > Delete Aggregate** можно вручную удалить те или иные типы объектов (агрегатов). Указанные в списке типы описаны в модели данных и появляются на этой вкладке после загрузки модели в систему.

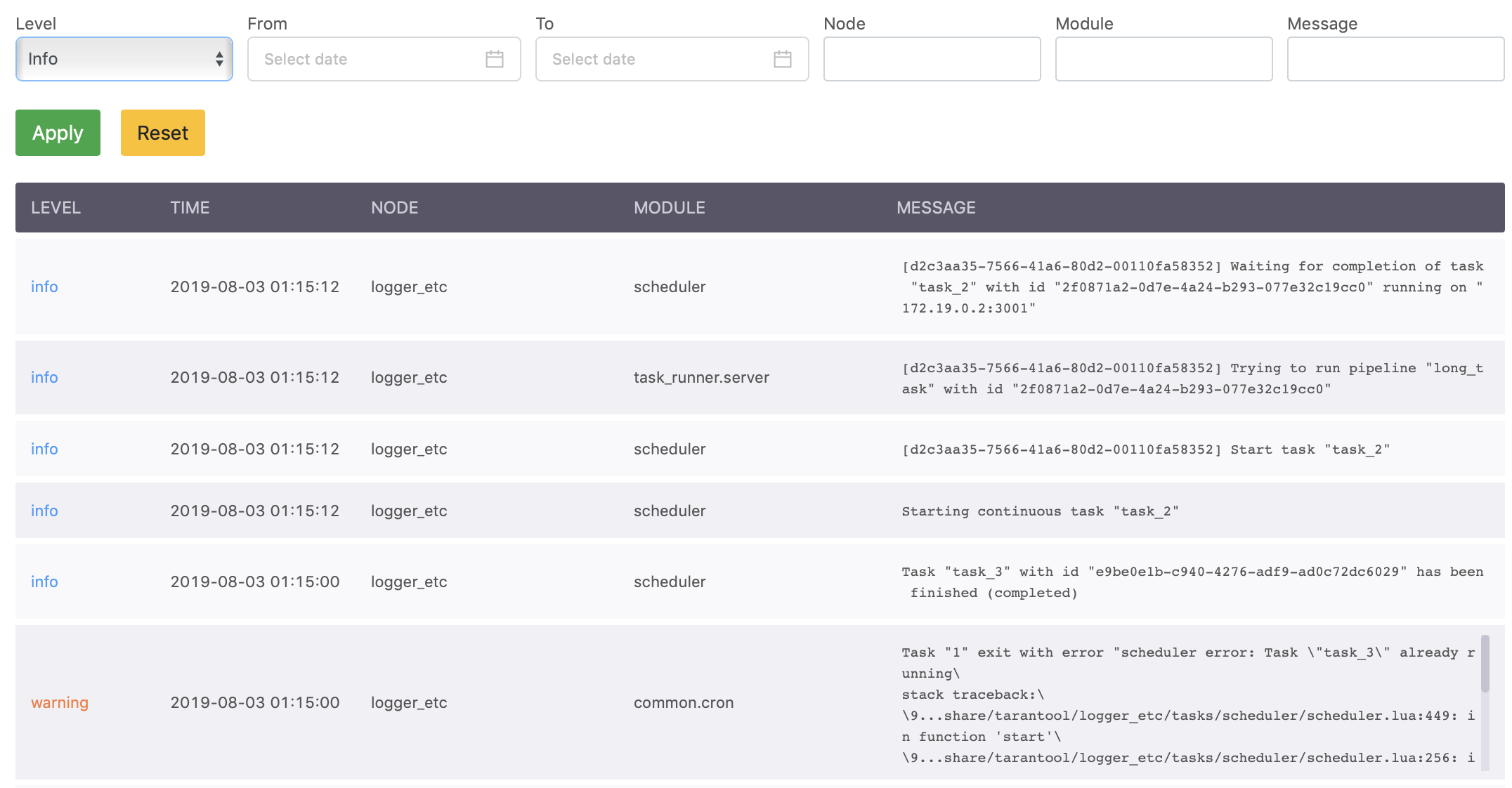


При удалении определенного типа также удаляются из storage все объекты указанного типа вместе с зависимостями (объекты логических типов «Entity» и «ValueObject»). При этом если объект типа «Entity» используется другим агрегатом, то он не удаляется. Подробнее про логические типы объектов см. раздел [«Разработка доменной модели»](#datamodelobjects).

## 8. Логирование

Просмотр лога событий, связанных с бизнес-процессами, доступен через web-интерфейс на вкладке **Logger**.

Для этого в системе должна быть определена роль logger и настроена его [конфигурация](#loggerconfig).



Каждая запись в таблице лога предоставляет следующую информацию о событии:

* Level — уровень логирования. В лог записываются события уровней «Info», «Warning» и «Error»;
* Time — дата и время события в формате «yyyy-mm-dd hh:mm:ss»;
* Node — имя узла кластера, на котором произошло событие;
* Module — имя модуля системы, инициировавшего событие;
* Message — описание события. В начале записи приводится UUID исходного запроса, с которым объект, в отношение которого возникло событие, пришел в систему. В случае события уровней «Warning» и «Error» в описание также включен полный stack trace.

Для удобства поиска записей в логе можно использовать фильтры. Фильтры существуют для каждой из колонок таблицы. Возможно использовать несколько фильтров совместно.

При заходе на страницу в таблице лога отображаются не более 100 записей. Если нужно увеличить количество записей, выводимых на экран, нажмите кнопку **Show more** внизу под таблицей.

Также возможно выгрузить все текущие записи лога в виде файла в формате .txt. Для этого нажмите кнопку **Save** внизу под таблицей.

Полный лог всех событий пишется средствами операционной системы и доступен при помощи системной утилиты journalctl:

journalctl -u <instance\_name>

Также полные логи для всех инстансов кластера можно получить при помощи скрипта expirationd tdgctl.py:

./tdgctl.py logs

См. подробнее про [формат данной команды](#logscommand).

## 9. Репликация объектов

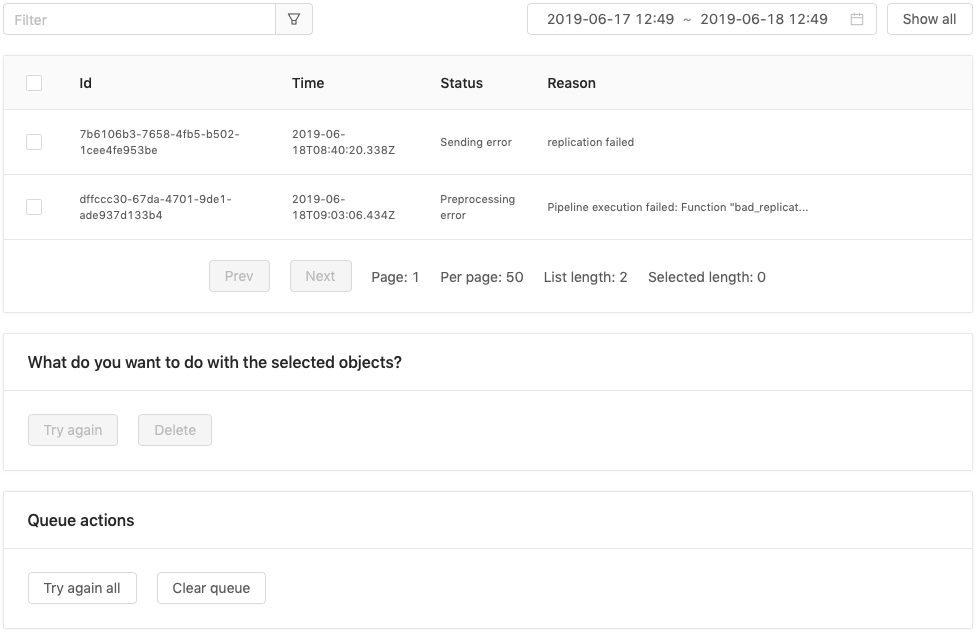
Механизм репликации объектов позволяет отправлять объекты во внешние системы в нужном формате.

Для работы репликации в системе должна быть определена роль output\_processor и настроена [конфигурация роли](#output_processor).

После успешной обработки на роли input\_processor объект направляется в хранилище (роль storage) с определенным ключом маршрутизации. Если в конфигурации системы для данного ключа предусмотрена репликация, объект также отправляется в очередь репликации. Далее объект проходит так называемый preprocessing — обрабатывается в пайплайне, указанном [в конфигурации](#output_processor) для роли output\_processor, и отправляется во внешнюю систему при помощи роли connector, где уже определен endpoint внешней системы.

Также возможно настроить репликацию определенных типов объектов, попавших в [ремонтную очередь](#repair-queue). Это тоже настраивается [в конфигурации](#section-repair-queue) системы.

Если во время репликации объекта произошла ошибка, объект попадает в специальную *ремонтную очередь репликации*. Ее функционал идентичен [ремонтной очереди](#repair-queue), но различие в том, что ремонтная очередь репликации содержит объекты, которые не удалось реплицировать, а не объекты, которые не удалось сохранить. Администрировать объекты в ремонтной очереди репликации можно через web-интерфейс на вкладке **Output\_Processor**.



Информация об объектах в этой ремонтной очереди, а также операции над ними (фильтрация, **Try again**, **Delete** и т.д.) аналогичны информации и операциям в основной [ремонтной очереди](#repair-queue). Отличаются только статусы объектов в этих двух очередях. В ремонтную очередь репликации объекты попадают в результате двух типов ошибок:

* ошибка на этапе обработки реплицируемого объекта (preprocessing) — статус «Preprocessing error»;
* ошибка на этапе отправки объекта во внешнюю систему — статус «Sending error».

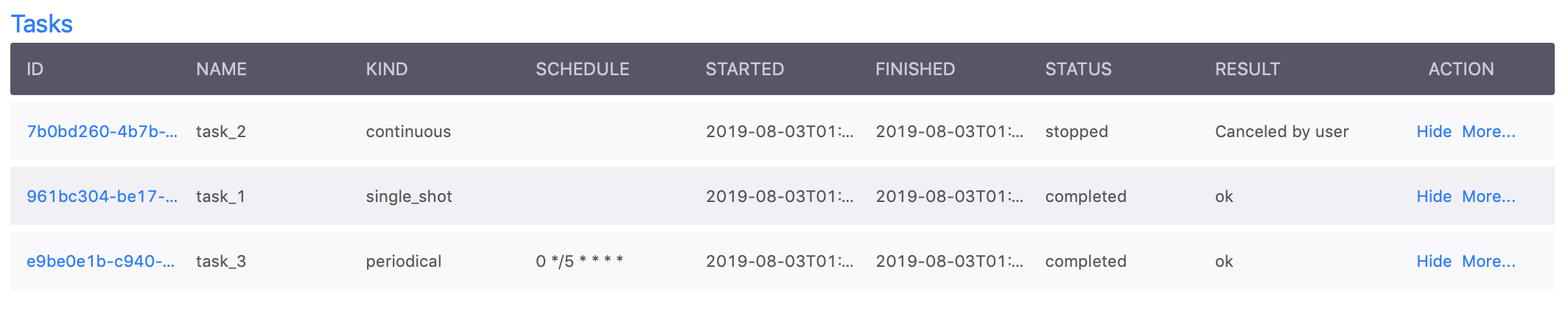
При повторной операции (**Try again**) над объектом его статус меняется на «In Progress». Если повторная операция успешна, объект переходит на следующий этап обработки или удаляется из ремонтной очереди (в зависимости от предыдущего статуса). Если повторная операция завершилась ошибкой, статус объекта меняется на «Rereplicated (Preprocessing error)» или «Rereplicated (Sending error)» (в зависимости от предыдущего статуса) и объект остается в ремонтной очереди репликации.

## 10. Задачи

Задачи суть те же пайплайны с набором функций, которые могут быть применены к сохраненным объектам или для любых других действий в системе (например, создание отчетов, инвалидация кэшированных данных и др.) и запущены в любое время, в т.ч. по расписанию.

Для выполнения задач в системе должны быть определены роли task\_runner и scheduler и настроена их [конфигурация](#section-tasks).

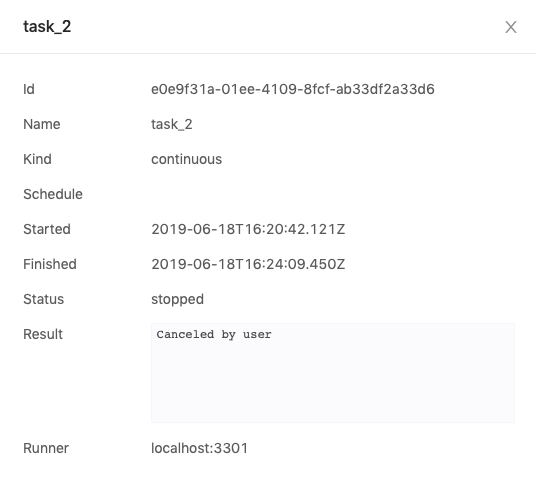
Отслеживать текущее состояние задач и управлять их выполнением можно через web-интерфейс на вкладке **Tasks**.



* **ID** — UUID экземпляра задачи.
* **Name** — Имя задачи.
* **Kind** — Вид задачи:
  + single\_shot — единоразовая задача;
  + continuous — непрерывно выполняемая задача;
  + periodical — задача, выполняемая по расписанию.
* **Schedule** — Расписание выполнения задачи. Актуально только для задач вида «periodical».
* **Started** — Дата и время старта экземпляра задачи.
* **Finished** — Дата и время окончания экземпляра задачи.
* **Status** — Текущий статус задачи:
  + did not start
  + pending
  + running
  + stopped
  + failed
  + completed
* **Result** — Сообщение о результате завершенной задачи (в статусе «stopped», или «failed», или «completed»).
* **Action** — Возможные действия для управления выполнением задач:
  + Start — запустить новый экземпляр неактивной задачи (задача в статусе «did not start» или «pending» — подсвечены зеленым в web-интерфейсе);
  + Stop — прекратить работу активного экземпляра задачи (в статусе «running»);
  + Hide — скрыть информацию об экземпляре задачи, завершившем свою работу (в статусе «stopped», или «failed», или «completed»).

Имя, вид и расписание выполнения задач определяются в [конфигурации системы](#section-tasks).

Информацию о конкретном экземпляре задачи можно получить в отдельном pop-up окне, которое выводится по клику на UUID задачи в колонке **ID**.



## 11. Конфигурация системы (config.yml)

config.yml — основной файл конфигурации системы, в котором задана логика и порядок обработки входящих запросов, а также настройки ролей.

Файл загружается в систему при [первоначальной конфигурации](#system-config-upload) после установки системы и настройки кластера.

Рассмотрим структуру файла и логику настроек на примере. Здесь в качестве справки приведены все возможные настройки. В реальной ситуации в конфигурации задаются только те настройки, которые необходимы для решения текущих бизнес-задач.

---  
types: {\_\_file: model.avsc}  
  
functions:  
 focus\_decode: {\_\_file: focus\_decode.lua}  
 focus\_routing: {\_\_file: focus\_routing.lua}  
 validate\_coupon\_payment: {\_\_file: validate\_coupon\_payment.lua}  
 focus\_classifier: {\_\_file: focus\_classifier.lua}  
 focus\_catchall: {\_\_file: focus\_catchall.lua}  
 notification: {\_\_file: notification.lua}  
 single\_task: {\_\_file: single\_task.lua}  
 long\_task: {\_\_file: long\_task.lua}  
  
pipelines:  
 t\_connect\_input\_handle:  
 - focus\_routing  
 coupon\_payment\_handle:  
 - focus\_decode  
 - validate\_coupon\_payment  
 focus\_catchall\_handle:  
 - focus\_decode  
 - focus\_catchall  
 focus\_classifiers:  
 - focus\_classifier  
 notification\_filter:  
 - notification  
 single\_task:  
 - single\_task  
 long\_task:  
 - long\_task  
  
connector:  
 input:  
 - name: soap  
 type: soap  
 wsdl: {\_\_file: Connect.wsdl}  
 handlers:  
 - function: Connect  
 pipeline: t\_connect\_input\_handle  
  
 - name: http  
 type: http  
 pipeline: t\_connect\_input\_handle  
  
 - name: kafka  
 type: kafka  
 brokers:  
 - localhost:9092  
 topics:  
 - orders  
 - items  
 group\_id: kafka  
 token\_name: kafka\_token  
 pipeline: t\_connect\_input\_handle  
  
 output:  
 - name: to\_input\_processor  
 type: input\_processor  
  
 - name: to\_external\_http\_service  
 type: http  
 url: http://localhost:8021/test\_json\_endpoint  
 format: json  
  
 - name: to\_external\_soap\_service  
 type: soap  
 url: http://localhost:8020/test\_soap\_endpoint  
  
 - name: to\_kafka  
 type: kafka  
 brokers:  
 - localhost:9092  
 topic: objects  
  
 - name: dummy  
 type: dummy  
  
 routing:  
 - key: input\_processor  
 output: to\_input\_processor  
  
 - key: external\_http\_service  
 output: to\_external\_http\_service  
  
 - key: external\_soap\_service  
 output: to\_external\_soap\_service  
  
 - key: dummy  
 output: dummy  
  
  
input\_processor:  
 classifiers:  
 - name: focus  
 pipeline: focus\_classifiers  
  
 routing:  
 - key: focus\_couponpayment  
 pipeline: coupon\_payment\_handle  
  
 - key: focus\_initiation  
 pipeline: focus\_catchall\_handle  
  
 - key: focus\_ratechange  
 pipeline: focus\_catchall\_handle  
  
 storage:  
 - key: focus\_couponpayment  
 type: CouponPayment  
  
output\_processor:  
 focus\_couponpayment:  
 pipeline: notification\_filter  
 output: to\_external\_http\_service  
 unclassified:  
 pipeline: notification\_filter  
 output: to\_external\_http\_service  
  
repair\_queue:  
 on\_object\_added:  
 \_\_unclassified\_\_:  
 postprocess\_with\_routing\_key: unclassified  
  
logger:  
 rotate\_log: true  
 max\_msg\_in\_log: 500000  
 max\_log\_size: 10485760  
 delete\_by\_n\_msg: 1000  
  
tasks:  
 task\_1:  
 kind: single\_shot  
 pipeline: single\_task  
 task\_2:  
 kind: continuous  
 pipeline: long\_task  
 pause\_sec: 10  
 task\_3:  
 kind: periodical  
 pipeline: long\_task  
 schedule: "0 \*/5 \* \* \* \*"  
  
services:  
 test\_query:  
 doc: "test query"  
 function: query  
 return\_type: TestObject  
 args:  
 a: string  
  
expiration:  
 - type: CouponPayment  
 lifetime\_hours: 12  
 delay\_sec: 1800  
 keep\_version\_count: 5  
  
hard-limits:  
 scanned: 2  
 returned: 2  
  
frontend:  
 - name: myapp  
 files:  
 "index.html":  
 \_\_file: app/index.html  
  
test-soap-data: {\_\_file: test\_object.json}

### 11.1. Секции

В файле конфигурации могут быть настроены следующие секции (подсвечены в листинге файла выше):

* **types** — описание модели данных (типов объектов), которые будут сохранятся в системе. В качестве языка описания модели используется Avro Schema — см. подробнее раздел [«Разработка доменной модели»](#domainmodeldev). Как правило описание делается в отдельном файле с расширением .avsc, на который ссылаемся в этой секции.
* **functions** — описание функций, которые будут выполняться в конвейерах обработки объектов (секция **pipelines**). Для каждой функции указывается имя функции и исполняемая часть (Lua-код функции, который также обычно выносится в отдельный файл).
* **pipelines** — описание конвейеров обработки объектов (пайплайнов). Для каждого пайплайна указывается имя и входящие в него функции, заданные в секции **functions**.
* [connector](#connectorheader) — конфигурация роли connector. Настраиваются:
  + *input*
  + *output*
  + *routing*
* [input\_processor](#input_processorheader) — конфигурация роли input\_processor. Настраиваются:
  + *classifiers*
  + *routing*
  + *storage*
* [output\_processor](#output_processor) — настройки для роли output\_processor.
* [repair\_queue](#section-repair-queue) — настройки обработки объекта при попадании в ремонтную очередь.
* [logger](#loggerconfig) — настройки логирования для роли logger.
* [tasks](#section-tasks) — конфигурация задач, выполняемых при помощи ролей scheduler и task\_runner.
* **services** — конфигурация сервисов.
* **expiration** — настройки времени жизни объектов и ограничения количества версий объектов.
* **frontend** — конфигурация роли frontend для статического расширения пользовательского интерфейса.
* **test-soap-data** — настройка позволяет задать текст, который будет по умолчанию отображаться на закладке **Test** в web-интерфейсе. Может использоваться для удобства тестирования: в этой секции можно задать структуру тестового объекта в формате xml или json для имитации входящего запроса.

#### 11.1.1. connector

*input* - настройки для получения и первоначальной обработки (parsing) входящих запросов. Поддерживаемые варианты (параметр type):

* http — для запросов в формате JSON по HTTP
* soap — для запросов формате XML по SOAP
* kafka — интеграция с шиной данных Apache Kafka.

*output* - настройки end-point’ов для отправки объектов. Поддерживаемые варианты (параметр type):

* input\_processor — отправка объекта на роль input\_processor
* http — отправка объекта в формате JSON во внешнюю систему по HTTP
* soap — отправка объекта в формате XML во внешнюю систему по SOAP
* kafka — интеграция с шиной данных Apache Kafka
* dummy — для использования тестовой заглушки.

*routing* — маршрутизация объекта для отправки через определенный *output*. *Output* определяется в зависимости от ключа маршрутизации, заданного параметром key.

#### 11.1.2. input\_processor

*classifiers* — определение пользовательского пайплайна, выполняющего классификацию объектов. Если данный параметр не задан, система использует классификатор, настроенный по умолчанию. В результате классификации объекту присваивается нужный ключ маршрутизации, по которому определяются дальнейшие действия по его обработке.

*routing* — маршрутизация объекта в определенный пайплайн обработки по ключу маршрутизации (параметр key).

*storage* — настройки сохранения объекта в роли storage по ключу маршрутизации (параметр key). Параметр type определяет тип бизнес-объекта (агрегата).

**Пример маршрута (порядка обработки) объекта**

1. После попадания в систему на роль connector, объект будет обработан в соответствии с настройками конфигурации *input* в секции **connector**.

В нашем примере мы видим, что для всех возможных вариантов *input* (JSON via HHTP, XML via SOAP, Kafka) конвейер обработки указан один и тот же: pipeline: t\_connect\_input\_handle.

2. В секции **pipelines** находим этот конвейер — он состоит из выполнения одной функции

pipelines:  
 t\_connect\_input\_handle:  
 - focus\_routing

3. В секции **functions** мы видим, что исполняемый код этой функции находится в файле focus\_routing.lua

functions:  
 ...  
 focus\_routing: {\_\_file: focus\_routing.lua}

Листинг кода:

local first = ...  
  
local ret = {obj = first, priority = 1, routing\_key = 'input\_processor'}  
  
return ret

Необработанный объект должен быть помещен в поле obj. Объекту будет присвоен ключ маршрутизации «input\_processor».

4. Дальнейший маршрут объекта определяется настройками *routing* в секции **connector**. Для объектов с ключом маршрутизации «input\_processor» это будет output: to\_input\_processor:

connector:  
  
 routing:  
 - key: input\_processor  
 output: to\_input\_processor

1. В секции **connector** находим данный *output*:

connector:  
  
 output:  
 - name: to\_input\_processor  
 type: input\_processor

Значение параметра type означает, что объект будет направлен на инстанс с ролью input\_processor.

6. При попадании на роль input\_processor объект прежде всего будет обрабатываться в соответствие с настройками из секции ниже:

input\_processor:  
 classifiers:  
 - name: focus  
 pipeline: focus\_classifiers

В указанном здесь пайплайне будет выполняться следующий код из focus\_classifier.lua:

local param = ...  
  
local type\_table =  
 {  
 ["Initiation"] = "focus\_initiation",  
 ["Coupon Payment"] = "focus\_couponpayment",  
 ["Exercise"] = "focus\_exercise",  
 ["Rate Change"] = "focus\_ratechange"  
 }  
  
local node = lom.get\_by\_path(param.obj, 'message.header.routedata.event\_type')  
  
if node == nil then  
 return param.obj  
end  
  
local event\_type = string.strip(node[1])  
  
param.routing\_key = type\_table[event\_type]  
  
return param

где lom — [[дописать]]

Допустим, что после обработки на данном этапе наш объект получит ключ маршрутизации «focus\_couponpayment».

7. Дальнейшая обработка объекта с таким ключом маршрутизации будет определяться следующими настройками в секции **input\_processor**:

input\_processor:  
  
 routing:  
 - key: focus\_couponpayment  
 pipeline: coupon\_payment\_handle  
  
 storage:  
 - key: focus\_couponpayment  
 type: CouponPayment

Это означает, что сначала объект будет обработан в еще одном пайплайне — coupon\_payment\_handle, а потом направлен на роль storage как объект типа «CouponPayment». Данный тип должен быть описан в модели данных, которая определена в секции

types: {\_\_file: model.avsc}

После успешной валидации по модели данных, объект будет сохранен на роли storage.

8. Если для данного ключа маршрутизации указаны еще какие-либо настройки в других секциях config.yml, объекты с этим ключом будут обработаны далее (см. пример про секцию [output\_processor](#output_processor)).

#### 11.1.3. output\_processor

Задается логика обработки объектов, которые будут реплицироваться во внешние системы. Для объектов с определенным ключом маршрутизации определяется:

* каким пайплайном будет обработан объект;
* посредством какого output он будет отправляться во внешнюю систему.

В рассматриваемом примере:

output\_processor:  
 focus\_couponpayment:  
 pipeline: notification\_filter  
 output: to\_external\_http\_service

Данные настройки означают, что объекты, которые на более ранних этапах обработки получили ключ маршрутизации «focus\_couponpayment», будут вначале обработаны функциями из пайплайна notification\_filter, а затем отосланы в соответствие с настройками указанного здесь *output* (смотрим эти настройки в секции **connector**):

connector:  
  
 output:  
 - name: to\_external\_http\_service  
 type: http  
 url: http://localhost:8021/test\_json\_endpoint  
 format: json

Аналогично определяется поведение роли output\_processor в отношение объектов с другими ключами маршрутизации.

#### 11.1.4. repair\_queue

repair\_queue:  
 on\_object\_added:  
 \_\_unclassified\_\_:  
 postprocess\_with\_routing\_key: unclassified

В данном примере настроен случай, когда определенные объекты, попавшие в ремонтную очередь, также должны быть реплицированы во внешнюю систему. В данном случае ранее на этапе классификации объект не удалось классифицировать, и ему было присвоено специальное значение ключа «\_\_unclassified\_\_», которым определяются неклассифицированные объекты.

При попадании такого объекта в ремонтную очередь, согласно настройкам postprocess\_with\_routing\_key: unclassified, объекту присваивается другой ключ маршрутизации «unclassified», и с этим ключом он направляется на роль output\_processor. Дальнейшая обработка объекта будет выполняться в соответствие с настройками в секции **output\_processor**:

output\_processor:  
 unclassified:  
 pipeline: notification\_filter  
 output: to\_external\_http\_service

#### 11.1.5. logger

В этой секции определяются настройки для роли logger. В нашем примере:

logger:  
 rotate\_log: true  
 max\_msg\_in\_log: 500000  
 max\_log\_size: 10485760  
 delete\_by\_n\_msg: 1000

где

* rotate\_log — указатель, осуществлять ли ротацию лога (значения true/false);
* max\_msg\_in\_log — максимальной количество сообщений, сохраняемых в логе;
* max\_log\_size — максимальный размер файла лога, в байтах;
* delete\_by\_n\_msg — количество одновременно удаляемых сообщений при ротации лога. При превышении значений параметров max\_msg\_in\_log или max\_msg\_log\_size наиболее старые n сообщений в логе удаляются за раз, что повышает производительность по сравнению с режимом, когда старые сообщения удаляются по одному.

#### 11.1.6. tasks

В этой секции настраивается конфигурация задач, выполняемых при помощи ролей scheduler и task\_runner. В нашем примере:

tasks:  
 task\_1:  
 kind: single\_shot  
 pipeline: single\_task  
 task\_2:  
 kind: continuous  
 pipeline: long\_task  
 pause\_sec: 10  
 task\_3:  
 kind: periodical  
 pipeline: long\_task  
 schedule: "0 \*/5 \* \* \* \*"

где

* task\_N — имя задачи;
* kind — вид задачи;
* pipeline — пайплайн, определяющий, что именно делается в рамках задачи;
* pause\_sec — пауза (в секундах) в выполнении задачи вида «continuous»;
* schedule — расписание выполнения для задач вида «periodical». Задается в формате cron. Используется расширенный синтаксис, в котором минимальной величиной является секунда:

\* \* \* \* \* \*  
| | | | | |  
| | | | | ----- День недели (0-6) (Воскресенье = 0)  
| | | | ------- Месяц (1-12)  
| | | --------- День (1-31)  
| | ----------- Час (0-23)  
| ------------- Минута (0-59)  
--------------- Секунда (0-59)

В примере выше значение параметра schedule: "0 \*/5 \* \* \* \*" означает, что задача будет запускаться периодически каждые 5 минут.

## 12. Разработка приложений

Систем-источников сложных бизнес-объектов может быть много, и они разрабатываются разными командами. Даже если системы работают с одними и теми же сущностями, их представление и структура, как правило, отличаются от системы к системе.

Чтобы унифицированно строить запросы по объектам из разных источников, система TDG приводит такие объекты к одному *каноническому* формату. Для задания формата используется специализированный высокоуровневый язык, позволяющий описывать структуру объектов, связи между ними и накладываемые на них ограничения.

Чтобы создать приложение для TDG, необходимо описать, разработать и организовать:

1. [Доменную модель](#domainmodeldev) (канонический формат объектов), используя соответствующий язык.
2. Само приложение, взаимодействующее с компонентами TDG и использующее [программный интерфейс репозитория](#repository-api) для управления объектами модели.
3. Версионирование модели (согласно ее [представлению](#model-storage)) и всех необходимых компонентов приложения для поддержания обратной совместимости.
4. [Тесты](#writing-tests) для проверки логики модулей TDG и приложения на соответствие требованиям.

Данное руководство описывает каждый шаг и содержит сведения, необходимые для их выполнения.

### 12.1. Разработка доменной модели

Чтобы приступить к описанию доменной модели, ознакомьтесь со следующими понятиями по порядку и изучите приведенный пример.

#### 12.1.1. Язык доменной модели

В терминологии системы TDG язык описания является *языком доменной модели*, а *домен* здесь – синоним понятия *предметная область*.

Язык доменной модели состоит из 2-х элементов:

* описания структуры объектов, и
* описания связей между объектами.

В качестве такого языка используется [Avro Schema](https://avro.apache.org). Он не переусложнен и распространен в сообществе и разрабатываемых им сторонних приложениях.

В стандарте Avro Schema есть два «контейнера», содержащих описания типов: *протокол* и *схема*. Оба – JSON файлы с расширением .avsc.

Приложение использует *схему* в качестве формата и понимает ее в виде массива типов:

[  
 {"name": "TypeA", "type": "record", ...},  
 {"name": "TypeB", "type": "record", ...}  
]

Каждый тип соответствует [стандарту Avro Schema](http://avro.apache.org/docs/1.8.2/spec.html), за исключением [расширений](#domainmodelext) для системы TDG. Расширения обратно совместимы, а модель, описанная с их помощью, должна успешно преобразовываться стандартными парсерами (синтаксическими анализаторами).

#### 12.1.2. Объекты модели: агрегат, сущность, значение

В дополнение к типичной для UML (Unified Model Language) терминологии [«агрегация и композиция»](https://www.visual-paradigm.com/guide/uml-unified-modeling-language/uml-aggregation-vs-composition/), в TDG есть три дополнительных понятия:

* **Агрегат (Aggregate)** – самостоятельный объект, имеющий идентичность.
* **Сущность (Entity)** – несамостоятельный объект, имеющий идентичность.
* **Значение (Value Object)** – несамостоятельный объект, не имеющий идентичности.
  + - Примечание
    - Значение здесь – отдельная единица моделирования (значение-объект), а не атрибут класса объекта.

Эти понятия – часть словаря [Domain Driven Design](https://lostechies.com/jimmybogard/2008/05/21/entities-value-objects-aggregates-and-roots/). Они описывают классы объектов с точки зрения *идентичности* и *принадлежности*.

*Идентичность* определяет, есть ли у объекта *жизненный цикл* во времени. Объект имеет идентичность, если остается тем же самым объектом, даже когда его свойства меняются.

Примером класса, объекты которого имеют идентичность, является класс *клиент* (бизнеса). Изменение имени клиента или его адреса не делает его другим клиентом – это тот же человек. С другой стороны, его *адрес* не имеет собственной идентичности, потому что смена улицы и дома сделает его другим адресом.

*Принадлежность* определяет, может ли объект существовать отдельно от другого. Объект самостоятелен, если его создание не требует существования других объектов или если можно *удалить* тот объект, который на него ссылается, в то время как самостоятельный объект будет *жив*.

Примером сущности, которая *не* может существовать отдельно, является *паспорт*. Паспорта выдаются только существующим людям и, даже если они были выданы «мертвым душам», полное удаление информации о «душах» потребует удаления информации об их паспортах.

Примером сущности, которая *может* существовать отдельно, является как сам клиент, так и его *договоры*. Если договоры или другие объекты, ссылающиеся на пользователя, будут удалены, существование объекта пользователя все еще будет иметь смысл. С другой стороны, если клиент пропал и был удален из системы, например, при компактизации данных или при их повреждении, существование его договоров тоже все еще будет иметь смысл, например, для построения финансовой отчетности.

Является объект агрегатом, сущностью или значением, важно не только для контроля жизненного цикла, но и для указания границ транзакционности и возможности ссылаться на объекты.

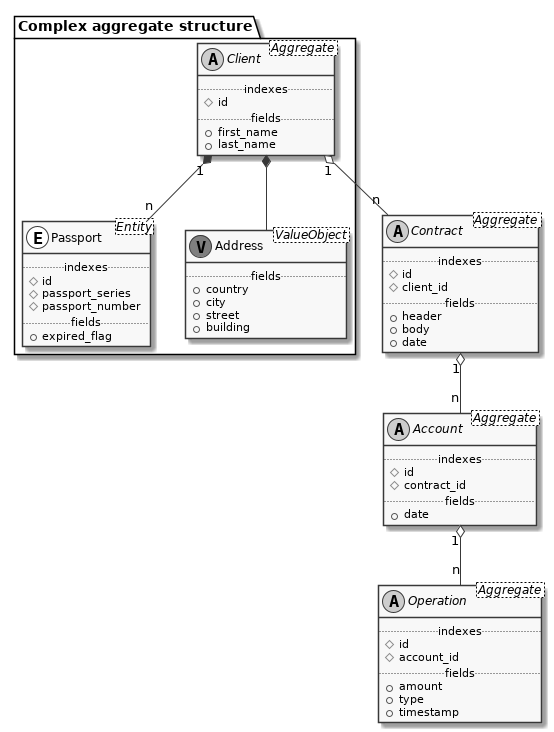
В пределах одного агрегата все его *подчиненные* сущности и значения можно обновить транзакционно и атомарно. Любая логика, работающая на уровне агрегата, может на это рассчитывать. Между агрегатами таких гарантий нет, то есть логика, обновляющая несколько агрегатов, является всегда *согласованной в конечном счете* (eventually consistent).

Также объекты могут ссылаться либо на объекты-сущности внутри своего агрегата-родителя, либо на другие агрегаты. Ссылки на сущности других агрегатов запрещены, так как сущности являются *деталью реализации* агрегата, и их данные всегда запрашиваются *через* объект агрегата.

#### 12.1.3. Пример описания на языке доменной модели

Предположим, в системе требуется хранить информацию о клиентах (Client), их паспортах (Passport), адресах (Address), договорах (Contract), счетах (Account) и операциях (Operation) по ним.

Рассмотрим следующую диаграмму объектов в качестве примера:



В примере:

* *Агрегатами* являются Client, Contract, Account и Operation. Информацию о каждом из них следует хранить вне зависимости от наличия других объектов – чтобы построить финансовую отчетность для любого временного среза.
* *Сущностью* – Passport. Информацию о паспортах хранить отдельно от клиентов не нужно, паспорта не существуют сами по себе. У каждого клиента может быть несколько паспортов и их состояние может меняться, например, срок действия – истекать. Поэтому каждый паспорт – отдельная сущность, зависимая от агрегата клиента.
* *Значением* – Address, так как он не обладает идентичностью. Система автоматически версионирует объекты, поэтому создавать массив для новых адресов не нужно, достаточно одного объекта-значения.

Из схемы также видны отношения:

* Между Client и Address, а также множеством экземпляров Passport, есть отношение владения.
* Объекты Client, Contract, Account и Operation существуют отдельно. Cвязи между ними – ссылочного типа.
* Одному и тому же агрегату разрешено находиться в отношениях агрегации с другими объектами.

Опишем данную структуру на языке доменной модели:

[  
 {  
 "name": "Passport",  
 "type": "record",  
 "logicalType": "Entity",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "passport\_series", "type": "string"},  
 {"name": "passport\_number", "type": "string"},  
 {"name": "expired\_flag", "type": "boolean"}  
 ],  
 "indexes": [  
 "id",  
 {  
 "name": "passport",  
 "parts": ["passport\_series", "passport\_number"]  
 }  
 ]  
 },  
 {  
 "name": "Address",  
 "type": "record",  
 "logicalType": "ValueObject",  
 "fields": [  
 {"name": "country", "type": "string"},  
 {"name": "city", "type": "string"},  
 {"name": "street", "type": "string"},  
 {"name": "building", "type": "int"}  
 ]  
 },  
 {  
 "name": "Client",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Клиент",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "first\_name", "type": "string"},  
 {"name": "last\_name", "type": "string"},  
 {"name": "passports", "type": {"type": "array", "items": "Passport"}},  
 {"name": "address", "type": "Address"}  
 ],  
 "indexes": ["id"],  
 "relations": [  
 {"name": "contracts", "to": "Contract", "count": "many",  
 "from\_fields": "id", "to\_fields": "client\_id"}  
 ]  
 },  
 {  
 "name": "Contract",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Договор",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "client\_id", "type": "long"},  
 {"name": "header", "type": "string"},  
 {"name": "body", "type": "string"},  
 {"name": "date", "type": {"type": "string", "logicalType": "Date"}}  
 ],  
 "indexes": [  
 "id",  
 "client\_id"  
 ],  
 "relations": [  
 {"name": "accounts", "to": "Account", "count": "many",  
 "from\_fields": "id", "to\_fields": "contract\_id"}  
 ]  
 },  
 {  
 "name": "Account",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Счет",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "contract\_id", "type": "long"},  
 {"name": "date", "type": {"type": "string", "logicalType": "Date"}}  
 ],  
 "indexes": [  
 "id",  
 "contract\_id"  
 ],  
 "relations": [  
 {"name": "operations", "to": "Operation", "count": "many",  
 "from\_fields": "id", "to\_fields": "account\_id"}  
 ]  
 },  
 {  
 "name": "Operation",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Операция",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "account\_id", "type": "long"},  
 {"name": "amount", "type": "double"},  
 {"name": "type", "type": "string"},  
 {"name": "timestamp", "type": {"type": "string", "logicalType": "DateTime"}}  
 ],  
 "indexes": [  
 "id",  
 "account\_id"  
 ]  
 }  
]

При описании мы использовали [расширения для TDG](#domainmodelext). Следующий раздел подробно описывает каждое расширение.

#### 12.1.4. Расширения доменной модели для TDG

Расширения дополняют спецификацию *Avro Schema* и позволяют воспользоваться функциональностью приложения.

TDG понимает следующие расширения:

##### 12.1.4.1. Расширение для агрегатов, сущностей и значений

Для указания признака агрегата, сущности или значения используется логический тип Avro Schema (Logical Type). Тип указывает приложению трактовать себя специальным образом.

В документации Avro Schema [сказано](http://avro.apache.org/docs/1.8.2/spec.html#Logical+Types), что логические типы можно использовать, например, для представления даты и времени. Наше приложение использует для [оперирования датами и временем](#datetimework) объекты DateTime, но на уровне сериализации и хранения они имеют строковый тип.

Поэтому даты, например, задаются так:

{ "type": "string", "logicalType": "Date"}

Стандарт Avro Schema не предписывает ничего по поводу допустимых значений logicalType, поэтому мы можем его использовать, чтобы придать типам дополнительный смысл.

TDG понимает следующие допустимые значения для logicalType для типов модели:

* Aggregate (агрегат),
* Entity (сущность),
* ValueObject (значение).

В [коде модели](#domain-model-code) мы задали типы всем классам объектов. Например, клиенту:

{  
 "name": "Client",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "fields": [...]  
 }

Если logicalType не указан, по умолчанию подразумевается ValueObject.

##### 12.1.4.2. Расширение для задания отношений между объектами

Явное задание отношений между объектами нужно для двух целей:

* Валидация внешних ключей при вставке объектов;
* Запрос связанных объектов через [graphql](http://graphql.org).

Для задания связи используется поле relations в теле описания класса объекта. Это поле не является стандартным, но игнорируется существующими парсерами Avro Schema. Связь – логическая конструкция. Связываемые поля (внешние ключи в терминологии SQL), должны быть объявлены на уровне классов.

Например, для связи между клиентом (Client) и его контрактом (Contract) требуются следующие поля:

{  
 "name": "Client",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "first\_name", "type": "string"},  
 {"name": "last\_name", "type": "string"},  
 {"name": "passports", "type": {"type": "array", "items": "Passport"}},  
 {"name": "address", "type": "Address"}  
 ],  
 "indexes": ["id"],  
 // Здесь должно быть поле "relations", формат которого описан ниже.  
},  
{  
 "name": "Contract",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Договор",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "client\_id", "type": "long"},  
 {"name": "header", "type": "string"},  
 {"name": "body", "type": "string"},  
 {"name": "date", "type": {"type": "string", "logicalType": "Date"}}  
 ],  
 "indexes": [  
 "id",  
 "client\_id"  
 ],  
}

Здесь:

* Первичный ключ клиента доступен через обращение к его классу – User.id.
* Внешний ключ находится в классе Contract и доступен аналогично – Contract.client\_id.

Чтобы сделать связь явной, определим поле relations в следующем формате:

"relations": [  
 {  
 // Все параметры обязательные.  
 "name": "<имя\_отношения>",  
 "to": "<класс\_объекта>",  
 "count": <"one"|"many">, // один к одному или один к многим  
 "from\_fields": <спецификация\_первичного\_ключа>,  
 "to\_fields": <спецификация\_внешнего\_ключа>  
 },  
 ...  
]

где:

* name – имя виртуального поля, через которое можно будет получить связанные объекты в graphql запросах.
* to – имя класса, с которым устанавливается связь.
* count – вид связи: один к одному или один к многим.
* from\_fields – спецификация поля, которое содержит первичный ключ.
* to\_fields – спецификация поля, которое содержит внешний ключ.

Спецификация обоих ключей (from\_fields и to\_fields) должна быть задана в следующем формате:

"index\_name" | "field\_name" | ["field\_name", "field\_name", "field\_name", ...]

То есть в полях from\_fields и to\_fields можно указывать имя индекса, имя поля (если оно одно) или список имен полей (если их больше).

Поле relations можно указать как с одной стороны отношения, так и с обеих. Односторонние отношения имеют смысл тогда, когда запрашивать данные в «обратном направлении» не требуется.

Полный пример задания отношения один к многим между Client и Contract с возможностью запроса данных в обоих направлениях:

[  
 {  
 "name": "Client",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "fields": [  
 {"name": "id", "type": "long"},  
 ...  
 ],  
 "indexes": ["id"],  
 "relations": [  
 {"name": "contracts", "to": "Contract", "count": "many",  
 "from\_fields": "id", "to\_fields": "client\_id"}  
 ]  
 },  
 {  
 "name": "Contract",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "client\_id", "type": "long"},  
 ...  
 ],  
 "indexes": [  
 "id",  
 "client\_id"  
 ],  
 "relations": [  
 {"name": "client", "to": "Client", "count": "one",  
 "from\_fields": "client\_id", "to\_fields": "id"}  
 ]  
 },  
]

##### 12.1.4.3. Расширение для задания ключей (индексов)

Для задания ключей используется поле indexes в описании класса. Так же как и relations, indexes не является частью спецификации, но не меняет поведения, регулируемого стандартом, а добавляет дополнительные ограничения на хранение и запросы.

Ключи описываются в следующем формате:

[<index1>, <index2>, <index3>, ...]

Где каждый ключ может быть:

* В виде строки:
* "<field\_name>"
* где field\_name – имя поля, по которому будет сделан ключ.
* Либо в виде словаря (для составных ключей):
* {  
   "name": "<index\_name>",  
   "parts": ["<field1\_name>", "<field2\_name>", ...]  
   [, "collation"="binary"|"case\_sensitivity"|"case\_insensitivity"]  
  }
* где:
  + index\_name – имя составного ключа, которое не должно совпадать с именами существующих полей.
  + field<X>\_name – имя одного из полей, по которому строится индекс.
  + collation – способ сравнения строк. По-умолчанию способ binary – бинарный 'A' < 'B' < 'a'. Значение case\_sensitivity включит регистрозависимое сравнение 'a' < 'A' < 'B'. Значение case\_insensitivity включит регистронезависимое сравнение 'a' = 'A' < 'B' и 'a' = 'A' = 'á' = 'Á'.

Для указания первичности ключа нет отдельного явного признака. Первичным ключом признается первый ключ в списке.

Полный пример задания ключей:

{  
 "name": "Passport",  
 "type": "record",  
 "logicalType": "Entity",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "passport\_series", "type": "string"},  
 {"name": "passport\_number", "type": "string"},  
 {"name": "expired\_flag", "type": "boolean"}  
 ],  
 "indexes": [  
 "id",  
 {  
 "name": "passport",  
 "parts": ["passport\_series", "passport\_number"]  
 }  
 ]  
}

Более сложный случай индексации – когда индекс делается по полю, присутствующему не в самом объекте, а в одном из его подобъектов. Такое возможно, если подобъект создается для логической группировки набора стандартных полей.

Например, представим, что операция по счету (Operation) в нашем примере производится по протоколу, требующему определенный заголовок. Создадим соответствующий объект-значение для него и сложный индекс в агрегате операции:

[  
 {  
 "name": "Header",  
 "type": "record",  
 "doc": "Заголовок операции",  
 "logicalType": "ValueObject",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "header\_body", "type": "string"}  
 ]  
 },  
 {  
 "name": "Operation",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Операция",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "header", "type": "Header"},  
 {"name": "account\_id", "type": "long"},  
 {"name": "amount", "type": "double"},  
 {"name": "type", "type": "string"},  
 {"name": "timestamp", "type": {"type": "string", "logicalType": "DateTime"}}  
 ],  
 "indexes": [  
 "id",  
 "account\_id",  
 {"name": "header\_id", "parts": ["header.id"]}  
 ]  
 }  
]

Header как объект-значение включается непосредственно в агрегат Operation, поэтому индекс может сослаться на поле из Header. Если бы Header был сущностью или агрегатом, модель не прошла бы валидацию.

##### 12.1.4.4. Расширение для задания распределения объектов по хранилищам

В случае распределенного хранилища данных агрегаты распределяются с использованием хеш-функции от первичного ключа объекта.

Для явного указания ключей для этой функции используется поле affinity с форматом:

"affinity": <index\_name>[, "affinity": index\_name, ...]

Директива может содержать только ключи входящие в первичный ключ.

Например, для распределенного хранения операций по счетам укажем:

{  
 "name": "Operation",  
 "type": "record",  
 "logicalType": "Aggregate",  
 "doc": "Операция",  
 "fields": [  
 {"name": "id", "type": "long"},  
 {"name": "account\_id", "type": "long"},  
 {"name": "amount", "type": "double"},  
 {"name": "type", "type": "string"},  
 {"name": "timestamp", "type": {"type": "string", "logicalType": "DateTime"}}  
 ],  
 "indexes": [  
 {"name":"pkey", "parts": ["id", "account\_id"]},  
 "account\_id",  
 ],  
 "affinity": "account\_id"  
}

Таким образом, операции одного и того же счета будут размещены физически на одном и том же хранилище.

#### 12.1.5. Работа с датой и временем

Внутри TDG для представления даты/времени используется строковый формат [ISO 8601](https://en.wikipedia.org/wiki/ISO_8601). Этот формат позволяет делать поля даты/времени индексируемыми, и, при этом, имеющими правильный порядок сравнения.

Все даты, поступающие в приложение, должны быть приведены к UTC.

Допустимые форматы записи:

* дата: YYYY-MM-DDZ
* время: HH:MM:SSZ
* дата/время с миллисекундами: YYYY-MM-DDTHH:MM:SS.sssZ
* дата/время с микросекундами: YYYY-MM-DDTHH:MM:SS.ssssssZ
* дата/время с наносекундами: YYYY-MM-DDTHH:MM:SS.sssssssssZ

Например: 2018-03-24T10:20:48Z.

Чтобы объявить поле типа Date, Time или DateTime, используйте механизм логических типов (logicalType) Avro Schema. Базовый же тип для этих полей – всегда строковый. Пример объявления поля даты/времени:

{"name": "timestamp", "type": {"type": "string", "logicalType": "DateTime"}}

### 12.2. Программный интерфейс репозитория

Приложение, взаимодействующее с TDG, использует программный интерфейс репозитория (выборки) как в качестве строительного блока для реализации GraphQL-запросов, так и во фрагментах кода, исполняемых, например, в тестовой среде («песочнице»).

Репозиторий поддерживает выборки разной сложности:

repository.get(type\_name, index\_name, value, options, context)  
  
repository.find(type\_name, filter, options, context)  
  
repository.map\_reduce(type\_name, filter, version, map\_fn, reduce\_fn, zero\_state, batch\_count)  
  
repository.update(type\_name, filter, updaters, options, context)  
  
repository.delete(type\_name, filter, options, context)

где:

* get – выборка по одному из ключей;
* find – выборка по полям объектов одного типа, включая несколько условий;
* map – сложная агрегация по кластеру;
* update – обновление объектов модели;
* delete – удаление объектов модели.

Функции get, find, update, delete поддерживают порядковую нумерацию страниц («пагинацию») с помощью параметров first и after.

Чтобы фильтровать объекты, запросы (все, кроме get) используют условия-предикаты (filter) – булевы выражения, синтаксис которых описан ниже.

#### 12.2.1. Синтаксис предикатов

В запросах предикаты (filter) записываются в виде:

{{left, comparator, right}, ...}

где:

* left и right – это левая и правая части выражения.
* Правая часть выражения (right) может содержать:
  + либо полный путь к полю объекта ($foo.bar), где имя поля начинается со знака $;
  + либо строковое или численное значение.
* Левая часть (left) может содержать только полный путь к полю.
* comparator – оператор сравнения: ">", ">=", "==", "<=" или "<".

Если в предикате несколько условий, по умолчанию они объединяются логической операцией and (конъюнкцией).

Примеры предикатов:

{{"$id", ">", 10}}  
{{"$id", ">", 10}, {"$id", "<", 100}}  
{{"$name", "==", "foo"}, {"$birth\_year", "==", 1990}}  
{{"$name", "==", "foo"}, {"$reg\_date", "==", {1990, 04, 23}}}

#### 12.2.2. Исторические данные

Запросы позволяют получить объекты, которые предшествуют или равны определённой версии. За это отвечает параметр options.version.

В запросах модифицирующих данные параметр options.version устанавливает создаваемую версию объекта.

#### 12.2.3. Оптимистичные блокировки

В случае конкурентной модификации объекта необходимо указать параметр options.only\_if\_version и модификация пройдет только в случае если данная версия объекта является последней и совпадает с указанной версией.

#### 12.2.4. Управление объектами модели

Приложение TDG может управлять объектами модели при помощи запросов на обновление (update) и удаление (delete).

##### 12.2.4.1. Запрос на обновление данных

Хранение данных в TDG представлено в виде «леса» из «деревьев». Деревья имеют свой уникальный идентификатор (ключевое поле), на основе которого они распределяются по хранилищам.

Поэтому запрос на обновление данных (update) выполняется в две стадии:

* исполнение запроса на каждом хранилище;
* сбор результатов на роутере.

**Синтаксис update-запроса**

Запрос на обновление объекта выглядит следующим образом:

repository.update(type\_name, filter, updaters, options, context)

где:

* type\_name – имя объекта для обновления;
* filter – список [условий-предикатов](#predicatsyntax) для выбора (фильтрации) объектов агрегата.
* updaters – список обновлений для объекта, состоящий из:
  + списка мутаторов {{mutator, path, new\_value}, ...}, где:
    - mutator – имя мутатора, например, set (устанавливает значение).
    - path – строковый путь до поля объекта с точкой-разделителем (.).
    - Путь до объекта(ов) массива должен включать индекс массива или символ \* для захвата всех подобъектов.
    - new\_value – новое значение.
* options – параметры для управления запросом:
  + first – количество элементов для обновления.
  + after – курсор пагинации на первый элемент.
  + version – версия объекта для основы.
  + dont\_skip\_deleted – использовать в поиске удаленные версии объектов
* context – контекст выполнения запроса
  + tenant – тенант

**Примеры update-запроса**

Для модели, рассмотренной в [примере](#domainexample):

* Чтобы обновить имя клиента с идентификатором 42, используйте следующий запрос:
* repository.update(  
   "Client",  
   {{"$id", "==", 42}},  
   {{"set", "first\_name", "John"},  
   {"set", "last\_name", "Doe"}})
* Если у того же клиента истек срок действия первого паспорта и необходимо обновить соответствующее поле expired\_flag, используйте следующий запрос:
* repository.update(  
   "Client",  
   {{"$id", "==", 42}},  
   {{"set", "passports.1.expired\_flag", "true"}}
* где .1. – индекс массива, содержащего экземпляры сущности Passport агрегата Client, т.е. первый паспорт клиента.

##### 12.2.4.2. Запрос на удаление объектов модели

Чтобы прозрачно и удобно исключить объект из потенциальных результатов запросов, используйте запрос на удаление (delete). Он помещает объект или его версию в «удаленное состояние» и исключает его из выборки.

**Синтаксис delete-запроса**

Чтобы удалить объект, используйте следующий запрос:

repository.delete(type\_name, filter, options, context)

где:

* type\_name имя объекта для удаления.
* filter – список [условий-предикатов](#predicatsyntax) для выбора (фильтрации) объектов агрегата.
* options параметры для управления запросом:
  + first количество элементов для удаления.
  + after курсор пагинации на первый элемент.
  + version версия объекта для копирования в удаленное состояние.
* context – контекст выполнения запроса
  + tenant – тенант

**Пример delete-запроса**

Чтобы удалить клиентов с именем «QWERTY» для модели из [примера](#domainexample), используйте следующий запрос:

repository.delete(  
 "Client",  
 {{"$first\_name", "==", "QWERTY"}})

### 12.3. Представление модели

При эксплуатации TDG объекты могут поступить в систему постфактум, а запросы к системе могут быть сделаны по функциональному срезу для определенной бизнес-даты.

Поэтому, чтобы поддерживать обратную совместимость при разработке приложения для TDG, нужно версионировать все, что непосредственно влияет на логику его работы:

* Avro схему модели (единый формат);
* конвейеры обработки входящих объектов;
* код Lua-функций, составляющих конвейеры и методы объектов модели;
* (потенциально) фабрики объектов;
* (потенциально) триггеры;
* (потенциально) описание сервисов.

Чтобы поддерживать историчность и эволюцию [модели](#domain-model), TDG представляет ее в едином формате – Lua-структуре, состоящей из примитивных типов. Ее легко преобразовать в JSON и обратно для передачи по сети.

Чтобы версионировать собственную структуру, система использует следующие форматы конфигурации для разных целей:

* [Внутренний формат](#model-storage-internal-config) – для передачи конфигурации по сети.
* [Экспортный формат](#model-storage-external-config) – для загрузки начальной модели при первичном старте системы, а также для загрузки конфигурации в тестах.
* [Экспортный формат бандла](#exportmodelformat) (комплекта) модели – для передачи разработчикам.

#### 12.3.1. Внутренний формат конфигурации

Узлы кластера передают конфигурацию по сети и хранят ее в следующем формате:

{  
 "functions": {  
 "function\_name": "function\_code"  
 ...  
 },  
 "pipelines": {  
 "pipeline\_name": ["function1\_name", "function2\_name", ...],  
 ...  
 },  
 "classifiers": {  
 "classifier\_name": "pipeline\_name/function\_name",  
 ...  
 },  
 "routing": {  
 "routing\_key": "pipeline\_name/function\_name",  
 ...  
 },  
 "storage": {  
 "routing\_key": "type\_name"  
 },  
 "schema": <avro\_schema>  
}

где:

* functions – набор функций, которые используются в классификаторах, маршрутизаторах и объектах модели;
* pipelines – конвейеры, в которых функции применяются к объекту по цепочке;
* classifiers – позволяют получить ключ маршрутизации для объекта;
* routing – функции, позволяющие менять ключ маршрутизации объекта;
* storage – хранилище для объектов с соответствующими ключами маршрутизации;
* function\_name – имя (идентификатор) Lua-функции;
* function\_code – код Lua-функции (исполняемый в «песочнице»);
* pipeline\_name – имя конвейера;
* classifier\_name – имя классификатора;
* routing\_key – строковый ключ маршрутизации.

Также в формате присутствует поле schema, формат которого описан в разделе [о доменной модели](#domain-model). В этом поле хранится структурное (а не строковое) ее представление.

#### 12.3.2. Внешний (экспортный) формат

Экспортный формат используется для загрузки начальной модели при первичном старте системы, а также для загрузки конфигурации в тестах.

Экспортный формат соответствует внутреннему, за исключением следующих полей:

* Вместо function\_code – function\_file\_name, файл, в котором находится код Lua функции.
* Вместо avro\_schema – avro\_file\_name, файл, в котором сериализована Avro схема модели.

Корневой объект сохраняется в файле config.yml; код функций – в соответствующие файлы с расширением .lua, рядом; а схема – в файл schema.avsc.

#### 12.3.3. Экспортный формат бандла моделей

Экспортный формат бандла (комплекта) может использоваться для удобной передачи разработчикам модели, загруженной в систему. Бандл – папка, содержащая в себе набор моделей, каждая в своей подпапке. Подпапки названы по бизнес-времени действия модели (а не по фактическому времени добавления).

Например:

bundle/  
├── 2018-01-03T18:56:00Z  
| ├── config.yml  
| ├── schema.avsc  
| ├── function\_1.lua  
| └── function\_2.lua  
└── 2018-03-05T00:41:00Z  
 ├── config.yml  
 ├── schema.avsc  
 └── function\_3.lua

Такой экспортный формат содержит только конфигурацию модели, но не содержит конфигурацию топологии системы или связи с внешними системами.

### 12.4. Разработка тестов

Тестирование приложений для системы TDG преследует две основные цели:

* Проверка логики модулей (*юнит-тесты*);
* Проверка продукта на соответствие требованиям (*интеграционные тесты*).

[Юнит-тесты](#unit-tests) обычно синтетические и предназначены для исчерпывающего покрытия критических частей кода и последующей автоматической проверки на регрессии. Самая большая польза от юнит-тестов – выявление ситуаций, в которых модуль может получить разнообразный неверный «ввод» от пользователя. В таких случаях юнит-тесты помогают убедиться, что модуль реагирует на неверный ввод корректно. Если система приняла неверные по форме/содержанию данные или, наоборот, не приняла верные данные (посчитав их неверными), то юнит-тесты – идеальное средство проверки подобных сценариев.

[Интеграционные тесты](#integration-tests) предназначены для проверки функциональности сложных элементов и для проверки на соответствие требованиям заказчика. Если поступает новое требование в формате пользовательской истории, то оно обязательно должно быть проверено интеграционным тестом, в котором должна быть сделана ссылка на требование.

Прежде чем разрабатывать собственные тесты, ознакомьтесь с существующими и запустите их.

#### 12.4.1. Запуск тестов

Чтобы запустить существующие юнит-тесты, в корне проекта:

1. Установите все зависимости при помощи скрипта:

* ./deps.sh
* Он установит в папку .rocks все модули, нужные для запуска.

1. Запустите юнит-тесты:

* tarantool unit.lua
* Команда выполнит по очереди все юнит-тесты из папки test/unit.

Чтобы запустить существующие интеграционные тесты:

1. Установите python3, pip и несколько python-библиотек. Весь список зависимостей приведен в файле requirements.txt в корне проекта. Чтобы установить их все глобально, используйте pip. Например:

* sudo pip3 install -r requirements.txt
  + - Примечание
    - Как альтернативу можно настроить [pipenv](https://docs.pipenv.org) и сделать для прогона тестов отдельную среду.

1. Запустите интеграционные тесты:

* pytest

#### 12.4.2. Разработка юнит-тестов

Юнит-тесты сгруппированы по подсистемам в подпапках test/unit. Чтобы тест был включен в общий прогон, имя файла должно начинаться с test\_ и иметь расширение .lua, например, test\_ddl.lua. Команда tarantool unit.lua найдет в упомянутой папке файлы с такими именами и автоматически исполнит их.

Юнит-тесты используют модуль [tap](https://tarantool.org/en/doc/1.7/reference/reference_lua/tap.html), который есть «на борту» Tarantool. Чтобы написать новый тест, используйте следующий код:

#!/usr/bin/env tarantool  
  
local tap = require('tap')  
local test = tap.test("validation")  
test:plan(5) -- замените цифру на количество тестов в модуле  
  
-- здесь будут тесты  
  
 os.exit(test:check() and 0 or 1)

Чтобы написать сами тесты, достаточно вызывать у объекта test функции проверки:

test:isnil()  
test:isstring()  
test:isnumber()  
test:istable()  
test:isboolean()  
test:isudata()  
test:iscdata()

Если в эти функции будут переданы данные, не соответствующие ожидаемым, тест будет прерван с ошибкой и поясняющим сообщением.

* + - Примечание
    - Чтобы быстро понять как писать юнит-тесты, лучше всего посмотреть в реализацию существующих.

Если в тестах требуется использовать box функции, используйте следующий код:

local tarantool = require('test.unit.tarantool').new()  
  
tarantool:start()  
  
-- здесь будут тесты  
  
local success = test:check()  
  
tarantool:stop()  
  
os.exit(success and 0 or 1)

#### 12.4.3. Разработка интеграционных тестов

Интеграционные тесты основаны на [pytest](https://docs.pytest.org/en/latest/), и «заточены» под тестирование логики приложения с возможным подключением логики кластера.

Для запуска каждого файла с тестами:

1. Сначала поднимается «с нуля» TDG или кластер из узлов с разными ролями.
2. Затем ко всем возможным узлам применяется соответствующая конфигурация.
3. Наконец, по очереди запускаются тест-кейсы.

Интеграционные тесты находятся в подпапках test/integration. В отличие от юнит-тестов, они сгруппированы не по подсистеме, а по логической «близости» тестов друг к другу. Основной критерий группировки – тесты в одной подпапке имеют одну и ту же стартовую конфигурацию.

Чтобы создать новую группу тестов:

1. Добавьте папку в test/integration и в ней сделайте подпапки config и data.
2. В config поместите все конфигурационные файлы для TDG и connector. При исполнении этой группы тестов, такие файлы будут автоматически загружены в качестве начальной конфигурации.
3. Дополнительно к папке config можно создать рядом папку data и положить туда тестовые данные, например, большие XML или JSON объекты. Это поможет не прописывать данные в коде теста в явном виде.
4. Создайте файл с именем, начинающимся с test\_ и расширением .py. Такие файлы автоматически включаются в прогон.
5. В файле используйте следующий код:

* #!/usr/bin/env python3  
    
  import pytest  
  import os  
    
  def test\_something(server, datadir):  
   # тут можно писать логику теста  
   assert something == something\_else  
    
  # тут можно добавить еще тестов

Здесь функция-тест test\_something имеет параметры server и datadir. Эти параметры называются «фикстуры» и они должны быть у каждой функции-теста. Фикстуры – это способ удобно использовать в тестах библиотечную функциональность и о них можно прочитать в [документации по pytest](https://docs.pytest.org/en/latest/fixture.html).

При указании в параметрах datadir, можно получить полный путь к папке, в которой лежат тестовые данные текущей группы тестов.

Когда pytest видит в параметрах тестовой функции server:

1. Он автоматически стартует TDG;
2. Применяет к нему конфигурацию;
3. И передает [объект-обертку](#wrapperapi) («враппер»), позволяющую удобно делать запросы (HTTP, SOAP, GraphQL и другие) к TDG.

#### 12.4.3.1. API объекта-обертки

Объект-обертка сервера («враппер») поддерживает запросы через HTTP, SOAP и GraphQL:

* server.post(path, data, json) – посылает post-запрос по пути path. Можно указать либо строку data для текстовых запросов, либо json для JSON запросов.
* server.soap(data) – послает SOAP-запрос, где data – текст запроса.
* server.graphql(query) – послает GraphQL-запрос, где query – текст запроса. В ответ функция либо бросает исключение при ошибке, либо возвращает объект с результатом. graphql для TDG разделён на схемы. Для доступа к данным schema = „default“ или не указывается. Для доступа к функциям администрирования schema = „admin“. Например запрос данных.
* server.graphql("""  
   query {  
   User(country:"USA") {  
   fullname  
   }  
   }  
  """)
* Пример вызова администрирования TDG, изменение модели:
* obj = server.post('/graphql', json={  
   "query": "mutation set\_model($model:String!) { model(model: $model) }",  
   "variables": {  
   "model": json.dumps(model)  
   },  
   "schema": "admin"  
  })
* server.cluster\_graphql(query) – посылает GraphQL-запрос для управления кластером. Например,
* obj = server.cluster\_graphql("""  
   {  
   servers {  
   uri  
   replicaset { roles }  
   }  
   }  
  """)

## 13. Доступ к данным

### 13.1. Язык запросов

Язык запросов системы основан на [GraphQL](http://graphql.org). При помощи него можно делать выборки объектов и ходить по связям.

#### 13.1.1. Выборка агрегатов

Все агрегаты (объекты логического типа «Aggregate» — см. подробнее в [описании модели](#datamodelobjects)) доступны для запроса по своему имени типа. Например, если в модели определен тип Book, объекты этого типа можно выбрать вот так:

{  
 Book(ISBN: "978-3-16-148410-0") {  
 title  
 year  
 author {  
 first\_name  
 last\_name  
 }  
 }  
}

Здесь параметр запроса ISBN соответствует индексу ISBN для типа Book. Проверка условия выполняется на полное совпадение.

В одном запросе можно также указывать несколько индексов, в таком случае запрос будет искать объекты, удовлетворяющие всем условиям одновременно (логическая операция И).

{  
 Book(year: 2018, issuing\_country: "USA") {  
 title  
 year  
 author {  
 first\_name  
 last\_name  
 }  
 }  
}

Для запросов такого рода доступны только индексированные поля. Фильтровать по обычным полям таким способом не получится.

Для запроса по мультиколоночному индексу используется массив значений. Допустим для Агрегата Book создан индекс fullname включающий в себя author.first\_name и author.last\_name, тогда запрос для вывода автора будет выглядеть так:

{  
 Book(fullname:['John', 'Snow']) {  
 title  
 year  
 }  
}

#### 13.1.2. Выборка со сравнением

Выборки поддерживают операции сравнения в виде суффиксов в именах индексов.

Например, для получения всех книг, выпущенных после 2008 года можно использовать запрос

{  
 Book(year\_gt: 2008) {  
 title  
 year  
 author {  
 first\_name  
 last\_name  
 }  
 }  
}

Поддерживаются следующие операторы сравнения

* \_gt (Greater Than) строго больше
* \_ge (Greater than or Equal) больше либо равно
* \_lt (Less Than) строго меньше
* \_le (Less than or Equal) меньше либо равно

Данные суффиксы поддерживаются и для мультиколоночных индексов. Предположим что есть индекс содержащий в себе year и month - year\_month. Тогда для получения книг выпущенных после июля 2008 года запрос будет такой

{  
 Book(year\_month\_gt: [2008, 07]) {  
 title  
 year  
 author {  
 first\_name  
 last\_name  
 }  
 }  
}

#### 13.1.3. Выборка агрегатов по связям

Для фильтрации связанных агрегатов используется тот же синтаксис, что и для выборки обычных агрегатов.

{  
 Book(year\_month\_gt: [2008, 07]) {  
 title  
 year  
 author(first\_name: "Anton") {  
 first\_name  
 last\_name  
 }  
 }

В этом случае будут выбраны все книги старше июля 2008 года, а авторы показаны только с именем Антон.

Данный вид запроса аналогичен SQL LEFT OUTER JOIN.

#### 13.1.4. Пагинация

Для пагинации мы используем метод с непрозрачными курсорами, похожий на тот, что описан [Здесь](http://graphql.org/learn/pagination/#pagination-and-edges).

Прототип запроса выглядит так:

MyType(first:2, after:$cursor)

Где:

* first указывает максимальное количество возвращаемых элементов (по умолчанию 10)
* after говорит с какого элемента продолжить выполнение запроса

В after как раз и передается «непрозрачный курсор». Непрозрачный курсор – это строка, о смысле которой пользователь не должен задумываться. Все, что нужно знать – это то, что используя эту строку, сервер может продолжить выполнение запроса с нужного места.

Каждый агрегат имеет специальное синтетическое поле cursor, доступное через graphql. В дизайне пагинации TDG было решено перенести cursor на уровень агрегата, что позволяет не вводить промежуточных уровней запроса edges и node, как это предлагается делать [В руководстве](http://graphql.org/learn/pagination/#pagination-and-edges).

Пример первого запроса:

{  
 CouponPayment(first: 2) {  
 header {  
 metadata {  
 object\_id  
 }  
 }  
 cursor  
 }  
}

Ответ:

{  
 "data": {  
 "CouponPayment": [  
 {  
 "header": {  
 "metadata": {  
 "object\_id": "1985222"  
 }  
 },  
 "cursor": "gaJwa5GnMTk4NTIyMg=="  
 },  
 {  
 "header": {  
 "metadata": {  
 "object\_id": "1985223"  
 }  
 },  
 "cursor": "gaJwa5GnMTk4NTIyMw=="  
 }  
 ]  
 }  
}

Теперь, чтобы продолжить получение следующей порции данных, нужно взять поле cursor с последнего объекта из результата (в данном случае – "gaJwa5GnMTk4NTIyMw==") и передать его в аргумент after:

{  
 CouponPayment(first: 2, after: "gaJwa5GnMTk4NTIyMw==") {  
 header {  
 metadata {  
 object\_id  
 }  
 }  
 cursor  
 }  
}

Для обратной пагинации необходимо использовать отрицательное число first. В данном случае система вернет предыдущие объекты относительно after. Обратная пагинация некольцевая, с помощью неё нет возможности получить последний объект множества выборки смещаясь от первого.

Пагинация также доступна для запроса по связям. Например:

{  
 Book(year\_month\_gt: [2008, 07]) {  
 title  
 year  
 author(first\_name: "Anton" first: 2) {  
 first\_name  
 last\_name  
 }  
 }

#### 13.1.5. Ограничения запросов

Для контроля нагрузки на сервер сделаны следующие ограничения запроса.

* запрос не должен проходить больше scanned строк;
* запрос не должен возвращать больше returned строк.

Данные ограничения превентивно требуют качественного написания запросов, являясь по сути аналогом профилирования медленных запросов.

Настройка параметров выполняется с помощью graphql запросов и мутаций.

Получить текущие значения ограничителей:

query power {  
 config {  
 hard\_limits {  
 scanned  
 returned  
 }  
 }  
}

Нарастить ограничители:

mutation gimme\_tha\_power {  
 config {  
 hard\_limits(scanned: 300000 returned:300000) {  
 scanned  
 returned  
 }  
 }  
}

## 14. Приложение

### 14.1. Скрипт tdgctl.py

Скрипт tdgctl.py позволяет развернуть приложение в кластере и выполнить ряд операций по его администрированию. В репозитории скрипт находится в [https://gitlab.com/tarantool/tdg/tdg/tree/master/deploy/](https://gitlab.com/tarantool/imdg/imdg/tree/master/deploy/).

Для использования скрипта необходимо подготовить [файл конфигурации кластера](#clusteconfigprepare) в формате JSON.

#### 14.1.1. Общий формат

tdgctl.py [-h] [-c <файл\_конфига>] [-v] <команда> [<опции>]

Команды:

* deploy — развертывание (deployment) инстансов;
* rm — удаление всех инстансов;
* logs — выгрузка логов всех инстансов;
* backup — резервное копирование данных всех инстансов;
* restore — восстановление данных из резервных копий;
* stop — остановка всех инстансов;
* start — старт всех инстансов.

Опции, общие для всех команд (далее в описании конкретных команд не упоминаются):

- h, -- help — вывод подсказки по использованию скрипта.

- c <файл\_конфига>, -- config <файл\_конфига> — передает скрипту файл конфигурации кластера в формате JSON. Если опция не указана, скрипт ищет по умолчанию файл с именем config.json в той же директории, где находится скрипт.

- v, -- verbose — вывод полного лога работы команды.

#### 14.1.2. deploy

Используется для развертывания инстансов.

Формат:

tdgctl.py deploy [-f] [-r] <файл\_образа>

где

<файл\_образа> — путь к файлу с образом установки в формате tar-архива.

- f, -- force — опция принудительной повторной установки (force re-deploy) на существующем кластере. Будет обновлена только функциональная часть, данные будут сохранены.

- r, -- roles — опция установки приложения в кластере с назначением ролей для инстансов. Для работы с этой опцией в файле конфигурации кластера должны быть указаны роли для каждого из инстансов. Например:

"servers":  
 [  
 {  
 "address": "172.19.0.2",  
 "username": "admin",  
 "instances":  
 [  
 {  
 "name": "core\_1",  
 "binary\_port": 3000,  
 "http\_port": 8080,  
 "memory\_mb": 128  
 "roles": ["connector", "input\_processor"]  
 },  
 {  
 "name": "storage\_1",  
 "binary\_port": 3001,  
 "http\_port": 8081,  
 "memory\_mb": 1024  
 "roles": ["storage"]  
 },

#### 14.1.3. rm

Используется для удаления всех инстансов кластера. Также будут удалены все данные.

Формат:

tdgctl.py rm [-y]

где

-y, --yes — опция удаления без запроса подтверждения.

#### 14.1.4. logs

Используется для выгрузки логов всех инстансов кластера.

Формат:

tdgctl.py logs [-d N]

где

-d N, --days N — опция выгрузки логов не старше N дней.

Для каждого из инстансов кластера формируется файл лога с именем <instance\_name>.log. При старте команды создается директория вида logs.yyyy-mm-ddThh:mm:ssZ, куда складываются все файлы логов, сформированные при данном выполнении команды (yyyy-mm-ddThh:mm:ssZ — дата и время на момент запуска команды).

#### 14.1.5. backup

Используется для резервного копирования данных со всех инстансов.

Формат:

tdgctl.py backup [-c]

где

-c, --clear — опция очистки директории для резервных данных перед началом резервного копирования.

Резервные данные сохраняются отдельно для каждого из инстансов в директории /var/lib/tarantool/<instance\_name>.checkpoint/.

В качестве резервных данных система сохраняет следующие файлы для каждого из инстансов:

* \*.snap — файлы [снимка данных (snapshot)](https://www.tarantool.io/ru/doc/2.1/book/box/data_model/#persistence) на момент резервного копирования.
* config.yml — файл [конфигурации системы из рабочей директории](#system-config-sync).
* config.backup.yml — резервная копия файла конфигурации системы.

#### 14.1.6. restore

Используется для восстановления данных из резервных копий инстансов.

Формат:

tdgctl.py restore

#### 14.1.7. stop

Используется для остановки всех инстансов.

Формат:

tdgctl.py stop

#### 14.1.8. start

Используется для старта всех инстансов.

Формат:

tdgctl.py start