Система TDG:  
руководство по техническому сопровождению и поддержке

## Оглавление

[Оглавление 2](#_Toc17119071)

[1. Введение 3](#_Toc17119072)

[1.1. Общие сведения о системе и руководстве 3](#_Toc17119073)

[1.1. Требования к персоналу 3](#_Toc17119074)

[2. Администрирование кластера 5](#_Toc17119075)

[2.1. Изменение топологии кластера (добавление инстансов) 5](#_Toc17119076)

[2.2. Балансировка данных 6](#_Toc17119077)

[2.3. Исключение инстанса из кластера 7](#_Toc17119078)

[2.4. Включение автоматического восстановления после отказа (Failover) 8](#_Toc17119079)

[2.5. Изменение мастера в наборе реплик 9](#_Toc17119080)

[2.6. Отключение набора реплик 10](#_Toc17119081)

[3. Настройки безопасности 11](#_Toc17119082)

[3.1. Пользователи и аутентификации 11](#_Toc17119083)

[3.2. Токен приложений 13](#_Toc17119084)

[3.3. Права доступа 14](#_Toc17119085)

[4. Ремонтная очередь 15](#_Toc17119086)

[4.1. Нотификации 17](#_Toc17119087)

[5. Управление бизнес-объектами 18](#_Toc17119088)

[5.1. Expiration 18](#_Toc17119089)

[5.2. Delete Aggregate 18](#_Toc17119090)

[6. Логирование 20](#_Toc17119091)

[7. Репликация объектов 22](#_Toc17119092)

[8. Задачи 24](#_Toc17119093)

[9. Тестирование и отладка 26](#_Toc17119094)

[9.1. Model 26](#_Toc17119095)

[9.2. Test 27](#_Toc17119096)

[9.3. Graphql 29](#_Toc17119097)

[9.4. Console 31](#_Toc17119098)

[10. Метрики 32](#_Toc17119099)

[11. Конфигурация системы (config.yml) 34](#_Toc17119100)

[11.1. Секции 37](#_Toc17119101)

[12. Устранение неисправностей 45](#_Toc17119102)

[13. Восстановление после сбоя 45](#_Toc17119103)

[14. Резервное копирование 45](#_Toc17119104)

[15. Приложение 45](#_Toc17119105)

[15.1. Скрипт tdgctl.py 45](#_Toc17119106)

## 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 процессов, обеспечивающих поддержание жизненного цикла системы, в том числе устранение неисправностей, выявленных в ходе эксплуатации, обновление системы, а также информацию о персонале, необходимом для обеспечения такой поддержки.

### 1.1. 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. Администрирование кластера

В данной главе описываются следующие операции по администрированию кластера:

* [изменение топологии кластера (добавление инстансов)](#topology)
* [исключение инстанса из кластера](#instance-expel)
* [включение автоматического восстановления после отказа (Failover)](#failover-enable)
* [изменение мастера в наборе реплик](#master-change)
* [отключение набора реплик](#replicaset-off)

### 2.1. Изменение топологии кластера (добавление инстансов)

При добавлении нового инстанса в набор реплик (новый или уже существующий) происходит следующее:

1. Кластер валидирует обновление конфигурации, проверяя доступность нового инстанса с помощью модуля membership. Все узлы в кластере должны быть рабочими, чтобы валидация была пройдена.

2. Новый инстанс ожидает, пока другой инстанс в кластере не получит обновление конфигурации и не обнаружит его. На этом шаге у нового инстанса еще нет своего UUID.

3. Как только инстанс понимает, что кластер знает о нем, инстанс вызывает функцию box.cfg и начинает работу.

Чтобы добавить в кластер новые инстансы:

1. Разверните необходимое количество новых инстансов Tarantool’а как описано в разделе по установке (см руководство по эксплуатации).

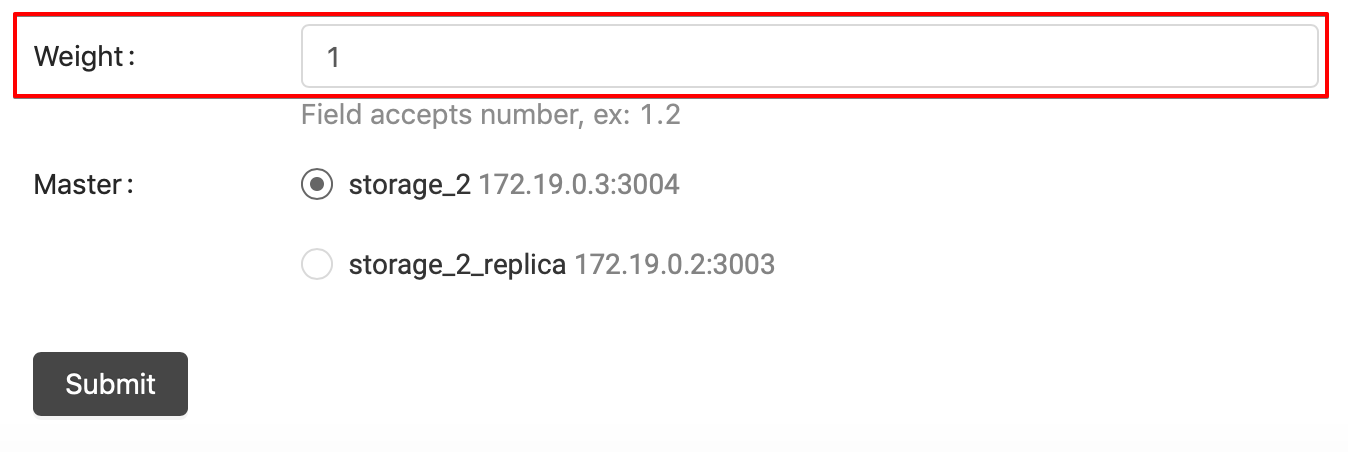
Если новые инстансы не появились в web-интерфейсе на вкладке **Cluster**, используйте функцию **Probe server** для проверки их доступности. Если они доступны, то появятся в списке.

2. В зависимости от того как вы хотите изменить топологию кластера, создайте с помощью вновь развернутых инстансов новые наборы реплик или включите новые инстансы в уже существующие наборы реплик. См. описание процедуры в разделе «Настройка кластера» (см руководство по эксплуатации).

3. Если вы добавляете инстанс с ролью storage, следует помнить, что вес (параметр Weight) такого инстанса по умолчанию устанавливается равным «0». Это определяется при инициализации модуля vshard, которая происходит во время первоначального развертывания кластера.

В данном случае нужно увеличить значение параметра Weight для того, чтобы система произвела балансировку данных, перенеся часть их на новый инстанс с ролью storage.

Для этого нажмите кнопку **Edit** у нужного набора реплик. В диалоговом окне **Edit Replica Set**, увеличьте значение параметра Weight и нажмите **Submit**, чтобы начать балансировку данных.



### 2.2. Балансировка данных

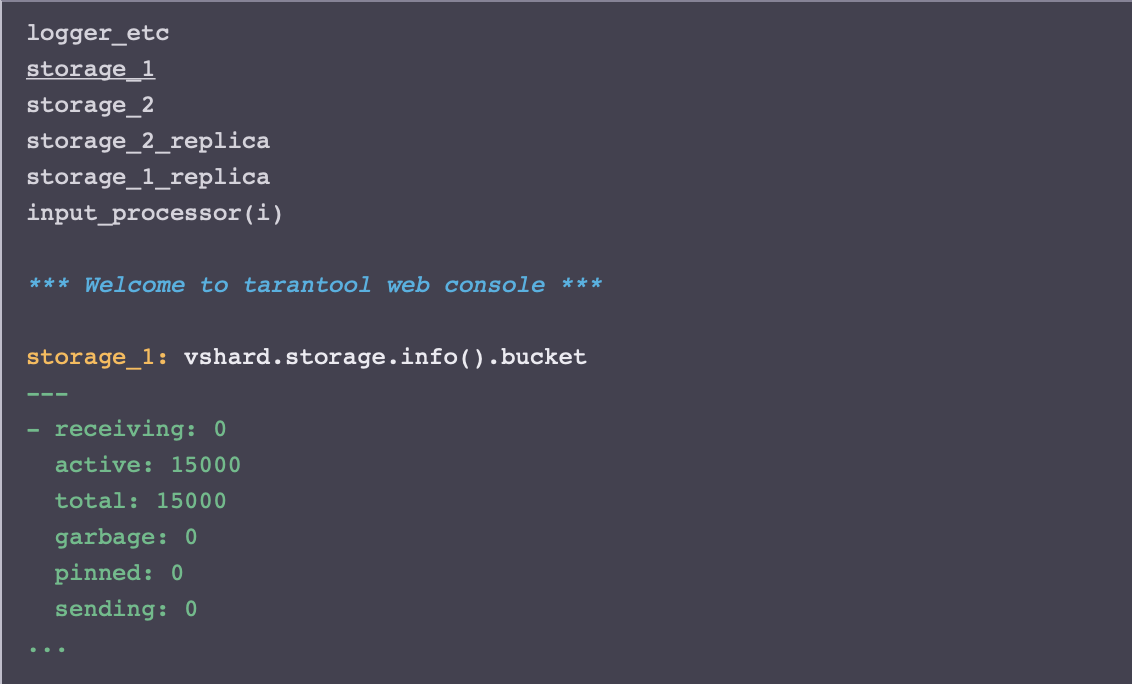
Балансировка данных (решардинг) запускается регулярно, а также после добавления в кластер нового набора реплик с ненулевым весом (параметр Weight). Для получения дополнительной информации см. [раздел о процессе балансировки](https://www.tarantool.io/ru/doc/2.1/reference/reference_rock/vshard/#rebalancing-process) в документации по модулю vshard.

Мониторинг процесса балансировки можно вести, отслеживая количество активных виртуальных сегментов (virtual buckets) на инстансах с ролью storage. Первоначально в новом наборе реплик нет активных сегментов. Через некоторое время фоновый процесс балансировки начинает переносить сегменты из других наборов в новый. Балансировка продолжается до тех пор, пока данные не будут распределены равномерно по всем наборам реплик.

Чтобы отслеживать текущее количество сегментов, подключитесь к нужному инстансу с ролью storage через консоль и выполните команду

vshard.storage.info().bucket

В web-интерфейсе администратора это можно сделать на вкладке **Console**.

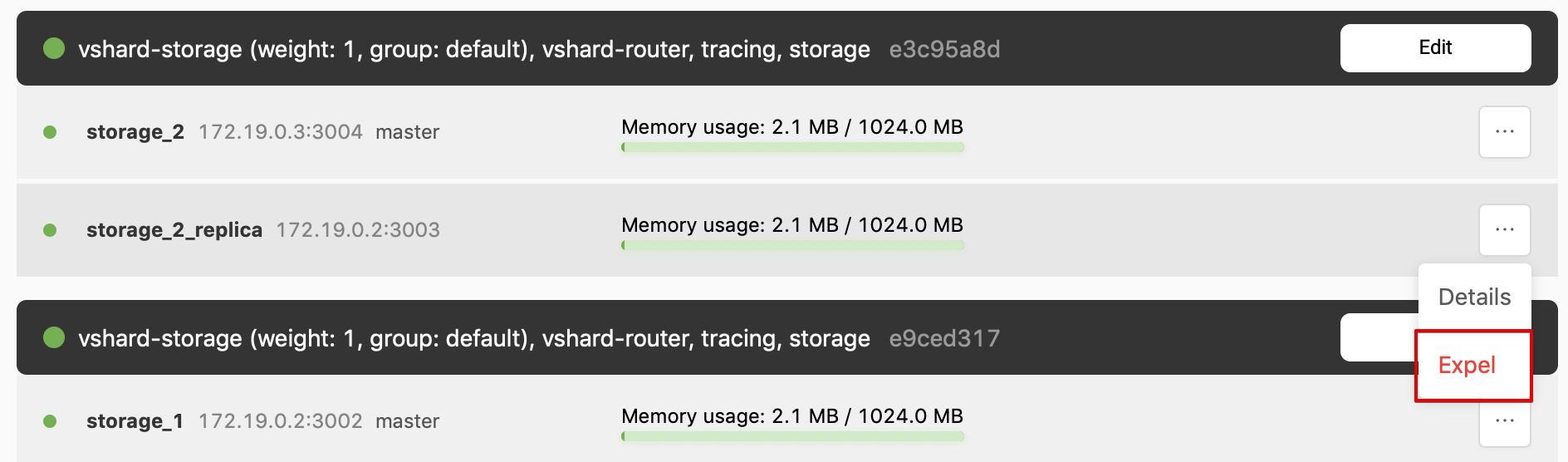


### 2.3. Исключение инстанса из кластера

Система позволяет исключить какой-либо инстанс из кластера. После того как инстанс будет исключен, остальные инстансы будут информированы об этом и не будут считать его членом кластера. Снова вернуть исключенный инстанс в кластер будет нельзя.

Для исключения инстанса из кластера:

1. В web-интерфейсе на вкладке **Cluster** для нужного инстанса нажмите **[…] > Expel**.



1. В окне подтверждения нажмите **OK**.

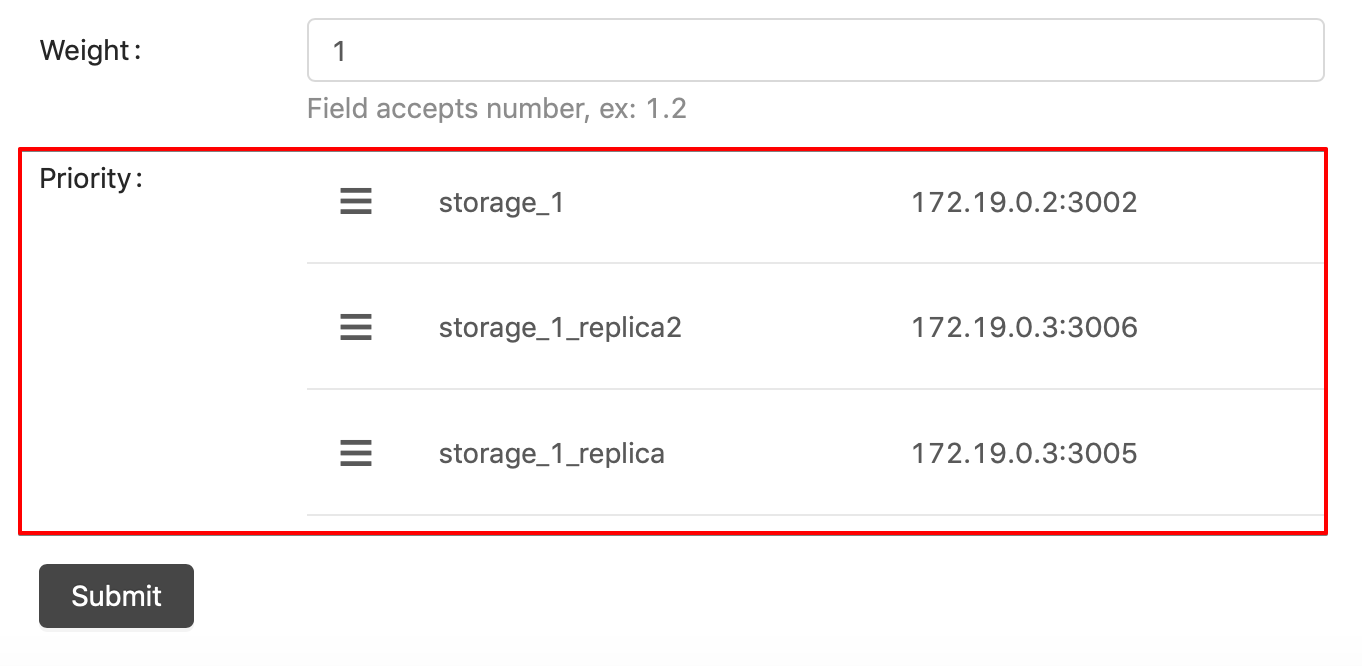
Инстанс больше не будет отображаться на вкладке **Cluster**.

### 2.4. Включение автоматического восстановления после отказа (Failover)

Если в кластере задана конфигурация «мастер-реплика» и включено автоматическое восстановление после отказа (failover), то при отказе мастера в каком-либо наборе реплик кластер автоматически выбирает следующую реплику из списка приоритетов и назначает ей роль активного мастера (read/write). Когда вышедший из строя мастер возвращается к работе, его роль восстанавливается, а назначенный ранее активный мастер снова становится репликой (read-only).

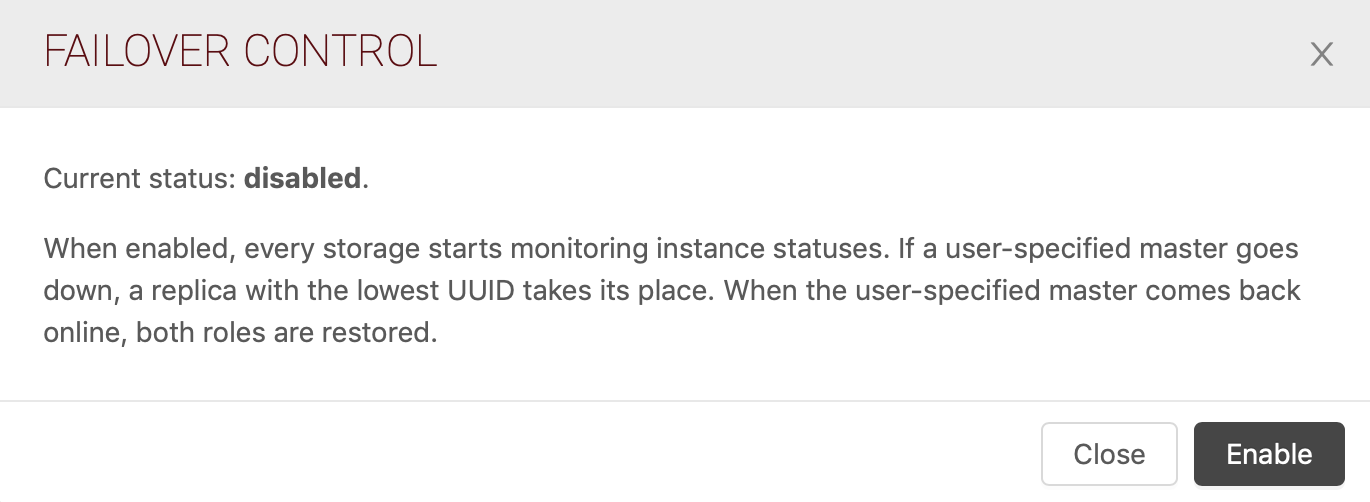
Чтобы установить приоритет инстансов в наборе реплик:

1. В web-интерфейсе на вкладке **Cluster**, нажмите кнопку **Edit** у нужного набора реплик.
2. В диалоговом окне отсортируйте, используя drag-and-drop, инстансы в списке в нужном порядке приоритета и нажмите **Submit**.

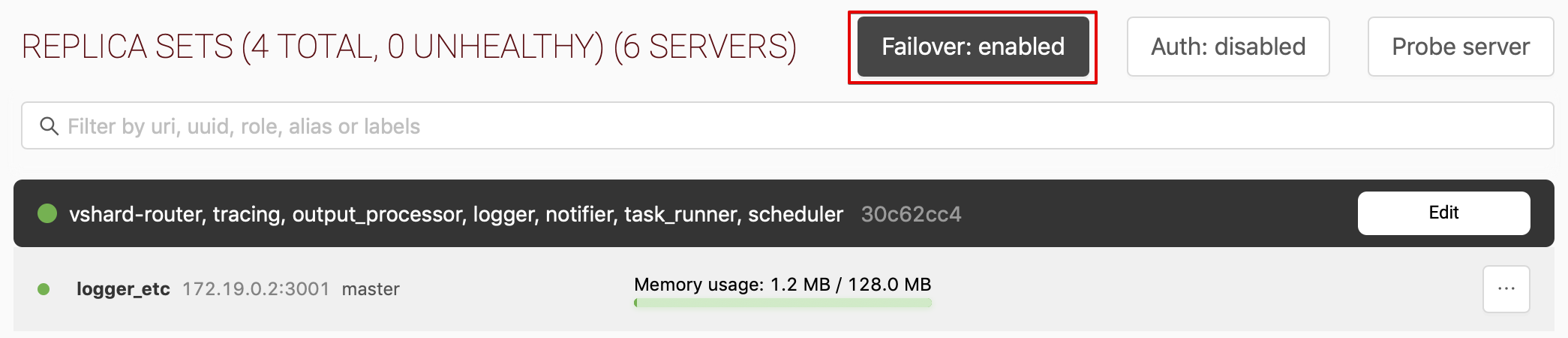


По умолчанию восстановление после отказа отключено, на что указывает статус на кнопке **Failover: disabled**. Нажмите эту кнопку для включения данной функции.

В диалоговом окне **FAILOVER CONTROL**, нажмите кнопку **Enable**.



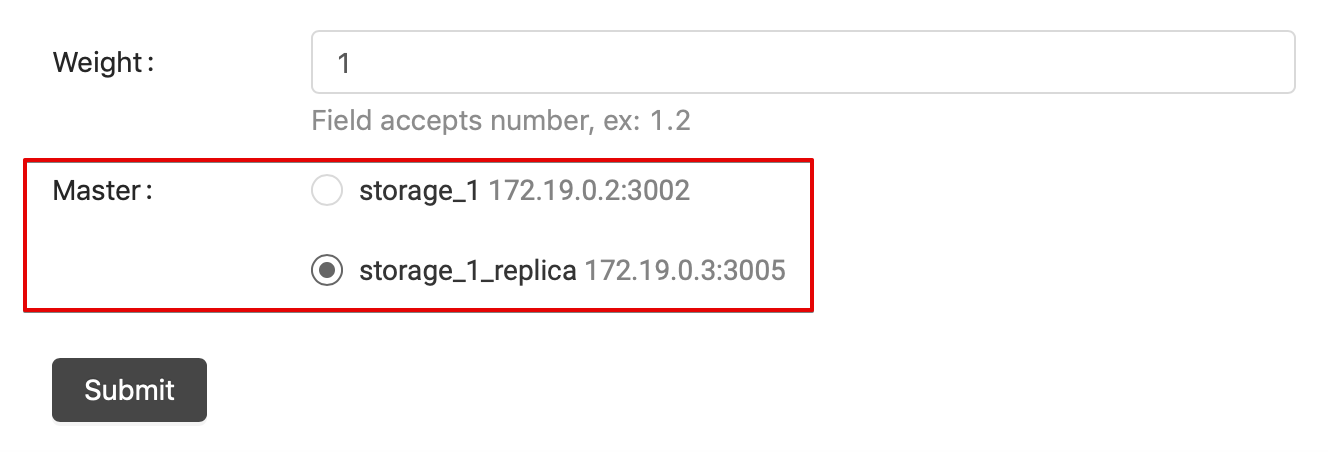
Статус функции восстановления после отказа изменится на **Failover: enabled**.



### 2.5. Изменение мастера в наборе реплик

Чтобы вручную изменить мастера в наборе реплик:

1. В web-интерфейсе на вкладке **Cluster**, нажмите кнопку **Edit** у нужного набора реплик.
2. В диалоговом окне выберите в качестве мастера другую ноду и нажмите **Submit**.

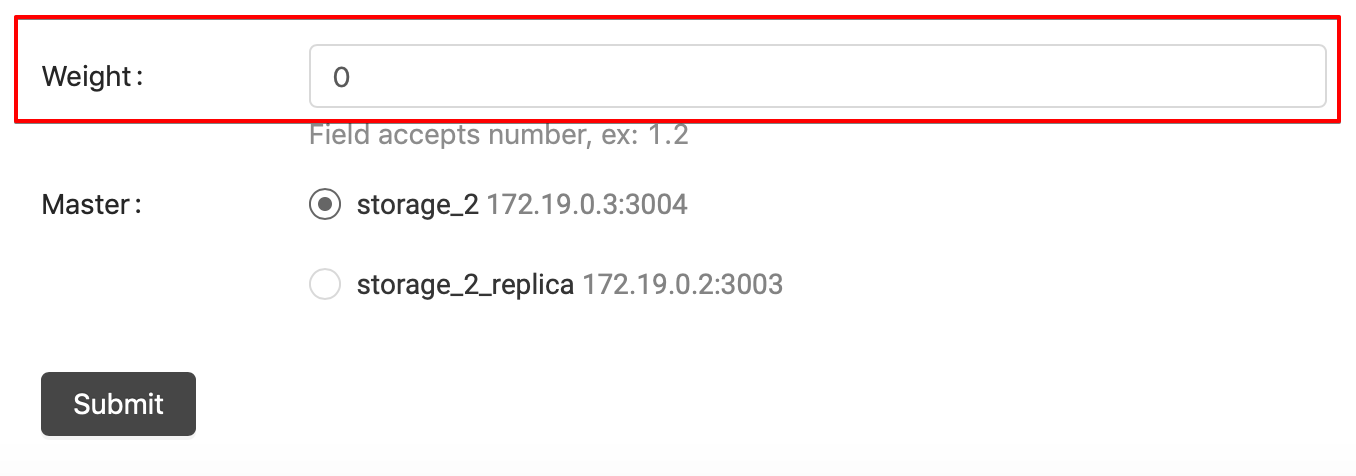


### 2.6. Отключение набора реплик

Под отключением набора реплик с ролью storage (например, для технического обслуживания) подразумевается перемещение всех его виртуальных сегментов в другие наборы реплик.

Чтобы отключить набор реплик:

1. В web-интерфейсе на вкладке **Cluster**, нажмите кнопку **Edit** у нужного набора реплик.
2. В диалоговом окне установите значение параметра Weight равным «0» и нажмите **Submit**.

* 

1. Подождите, пока процесс балансировки не завершит перенос всех виртуальных сегментов. Текущее количество сегментов в данном наборе реплик можно отслеживать как это описано в [разделе о балансировке данных.](#databalancing)

## 3. Настройки безопасности

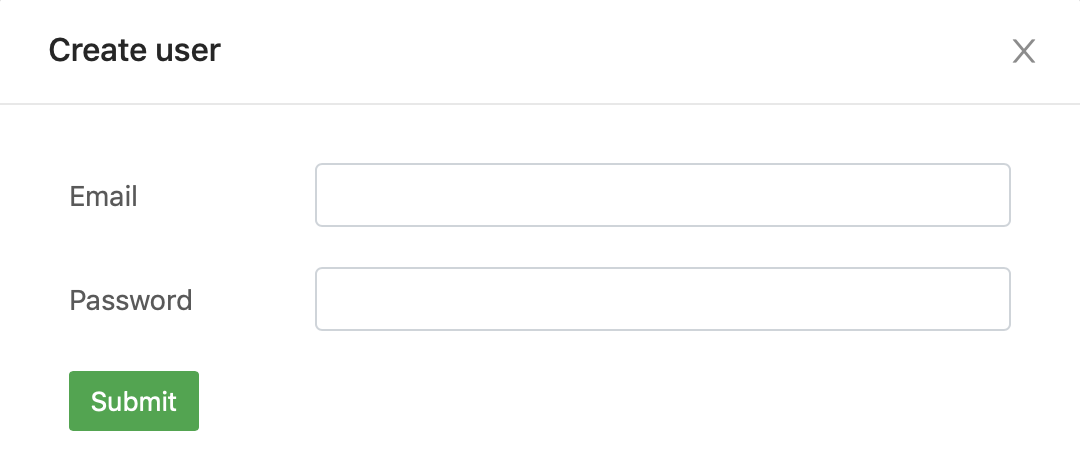
### 3.1. Пользователи и аутентификации

По умолчанию доступ к web-интерфейсу администратора не ограничен, и анонимный пользователь имеет доступ ко всем функциям и объектам.

Для системы в промышленной эксплуатации необходимо создать авторизованных пользователей, для которых в дальнейшем можно регулировать права доступа к тем или иным типам объектов, и включить режим обязательной аутентификации в системе.

#### 3.1.1. Создание пользователей

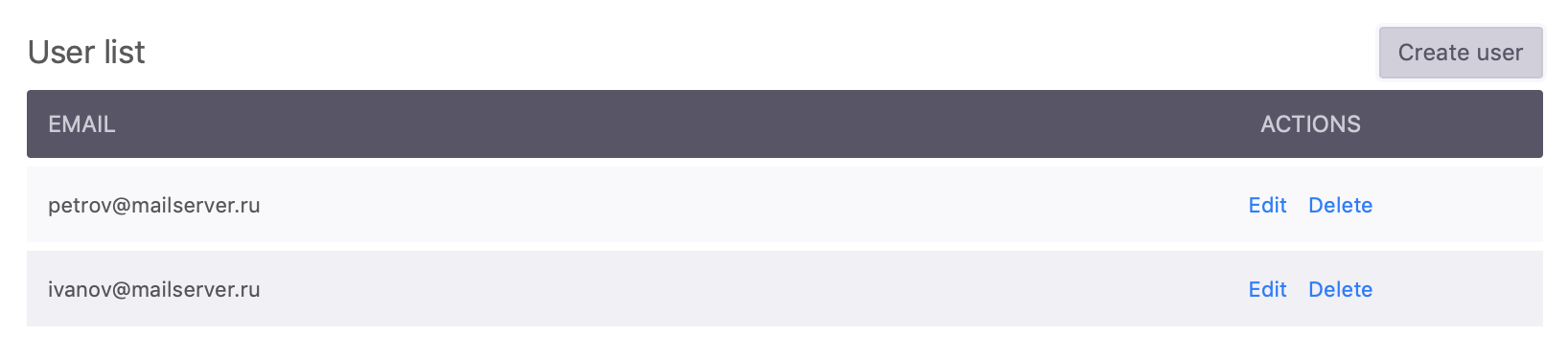
Для создания пользователя перейдите на вкладку **Settings > Users** и нажмите **Create user**. В диалоге создания пользователя укажите Email, который будет служить идентификатором пользователя, и пароль и нажмите **Submit**.



Созданный пользователь будет отображен в списке **User list**.

В дальнейшем администратор может

* создавать новых пользователей;
* редактировать данные текущих пользователей;
* удалять пользователей;
* [управлять правами доступа](#permissions) пользователя к объектам.

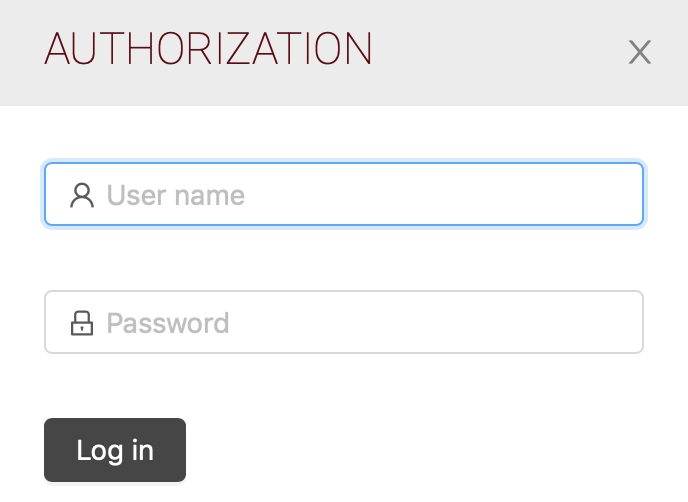


#### 3.1.2. Авторизация в системе

Для авторизации в системе первый раз (когда режим аутентификации еще выключен) нажмите кнопку **Log in** в правом верхнем углу в web-интерфейсе.



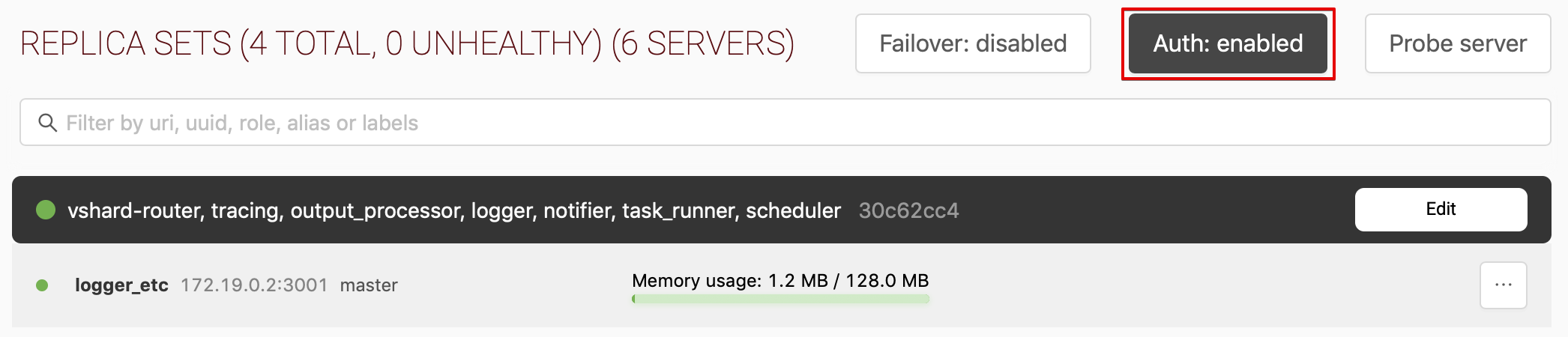
В форме авторизации введите имя пользователя, в качестве которого выступает Email, и пароль и нажмите **Log in**.



#### 3.1.3. Режим обязательной аутентификация

На вкладке **Cluster** кнопка **Auth: <mode>** включает и выключает режим обязательной аутентификации, где <mode> — указатель текущего режима: «disabled» (выключен) или «enabled» (включен).

Для включения режима обязательной аутентификации авторизуйтесь в системе, перейдите на вкладку **Cluster** и нажмите кнопку **Auth: disabled**. Индикация режима должна смениться на **Auth: enabled**.

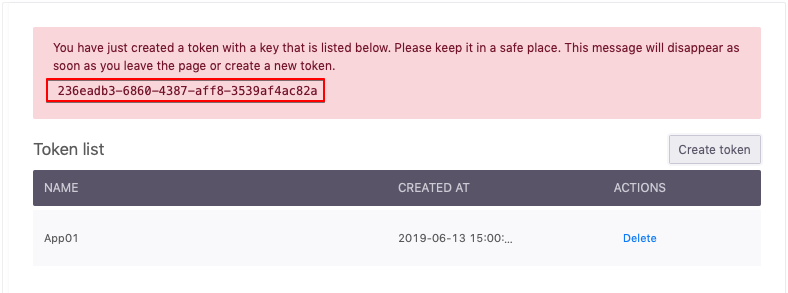


### 3.2. Токен приложений

Приложениям внешних систем для доступа к данным в системе TDG необходим авторизованный доступ, который организуется через токены приложений.

Общий процесс выглядит следующим образом: в web-интерфейсе администратор генерирует токен, назначает для него права доступа к объектам системы TDG и передает токен разработчикам внешней системы.

Генерация токена выполняется на вкладке **Settings > Tokens**, кнопка **Create token**. При генерации необходимо указать имя (ключ) токена, которое будет в дальнейшем идентифицировать его в системе. После генерации токен в явном виде будет доступен только в сообщении в web-интерфейсе:



* **Примечание**
* Сохраните сгенерированный токен в надежном месте. Сообщение с токеном исчезнет, как только вы покинете данную страницу или сгенерируете новый токен.

В дальнейшем сгенерированный токен является нередактируемым, его можно только удалить из системы (в web-интерфейсе: кнопка **Delete** в колонке **ACTIONS** для соответствующего токена).

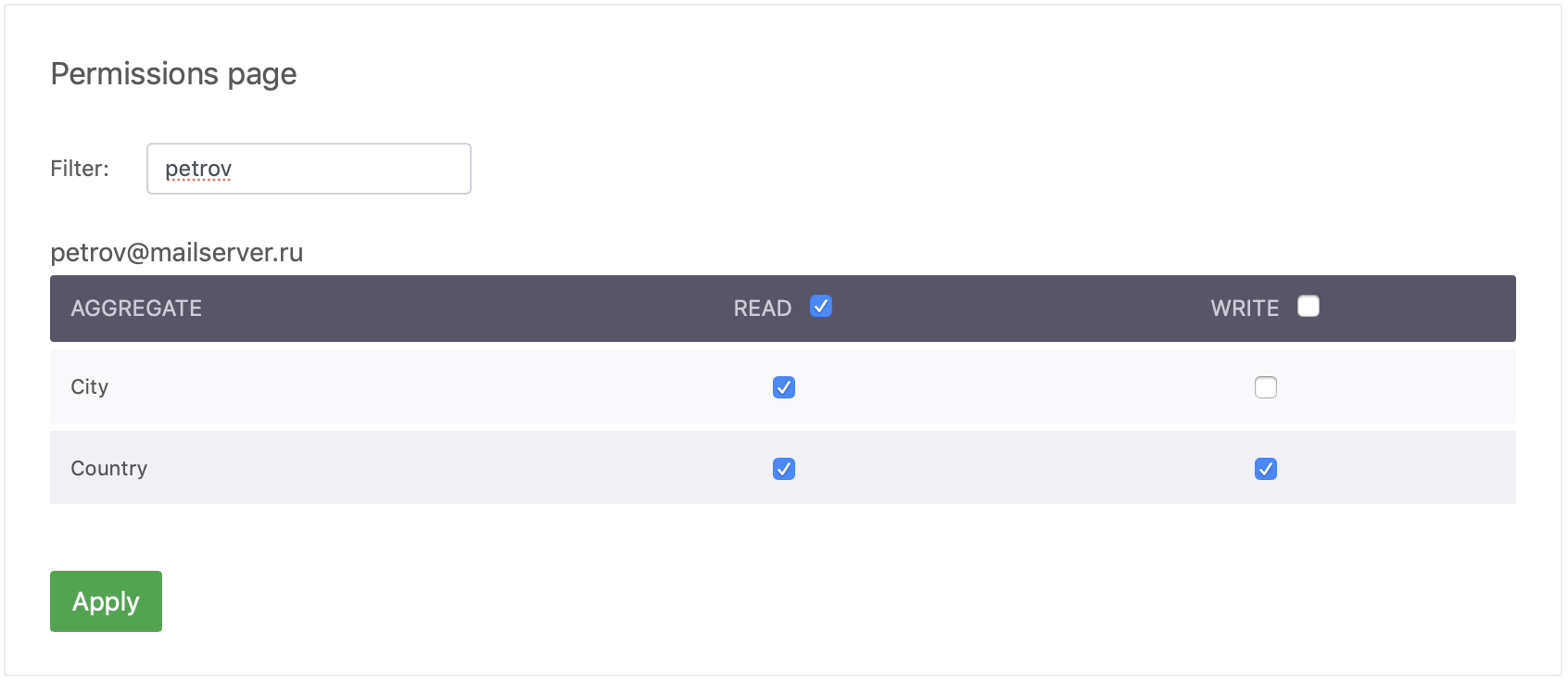
Настройка прав доступа токена к объектам аналогична настройке прав доступа для пользователей и описана в разделе [«Права доступа»](#permissions).

### 3.3. Права доступа

Система дает возможность установить для каждого пользователя или токена права на действия с бизнес-объектами (объект с логическим типом «Aggregate», далее — агрегат):

* Read — чтение (получение) информации о существующих объектах;
* Write — вставка новых объектов или обновление существующих объектов.

Установить права можно через web-интерфейс на вкладке **Settings > Permissions**. Устанавливать права нужно после того, как в систему загружена модель данных, в которой описаны агрегаты. Модель данных загружается при первоначальной конфигурации системы.



Чтобы установить права пользователя/токена:

1. [опционально] С помощью поля **Filter** отфильтруйте настройки для нужного пользователя или токена.
2. Для каждого из агрегатов отметьте галочками нужные права, Read и/или Write.
3. Примените настройки прав, нажав **Apply**.

Настройки прав возможно редактировать, когда это необходимо, — при появление новых пользователей или токенов, добавление новых агрегатов и т.д.

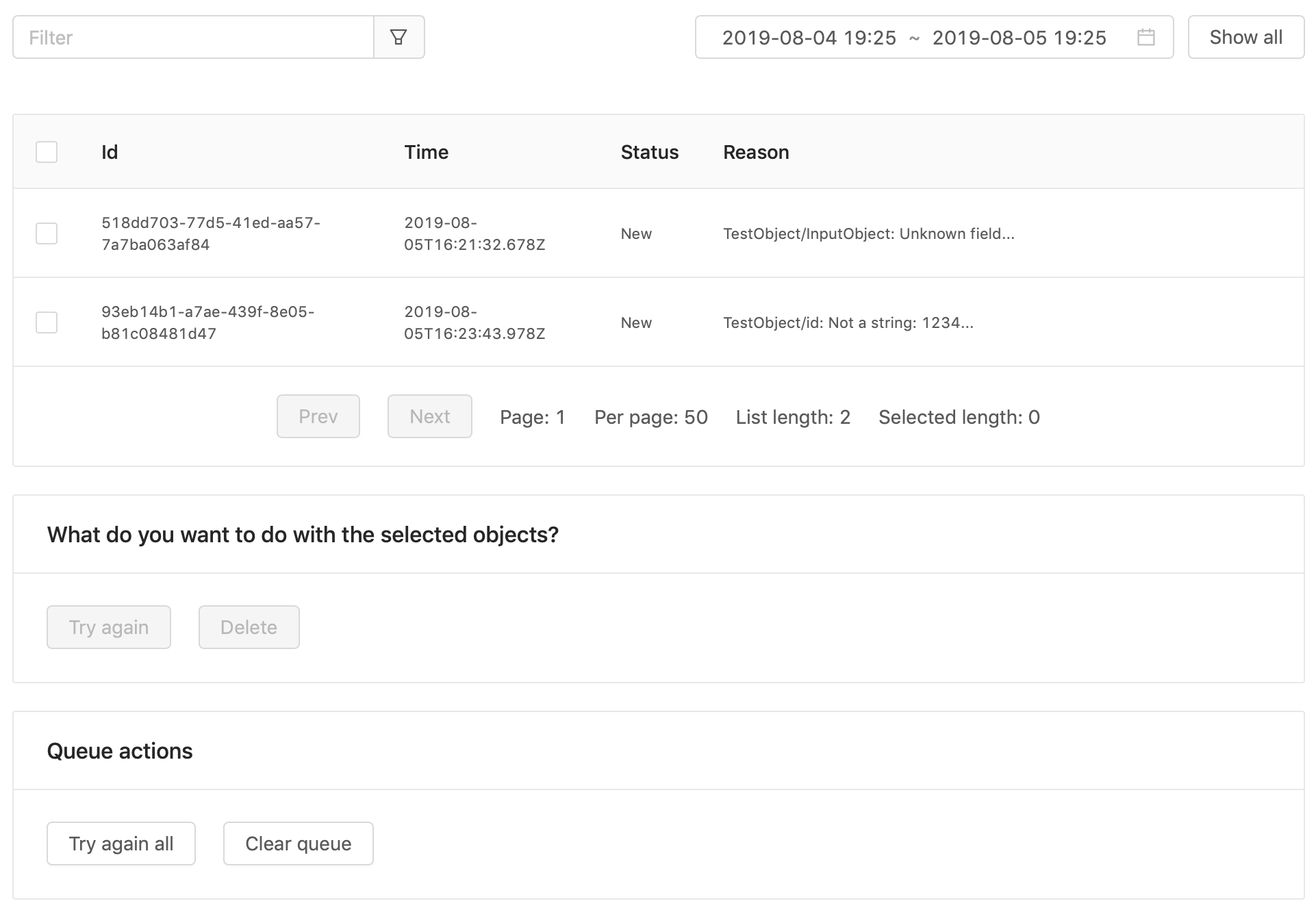
## 4. Ремонтная очередь

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:

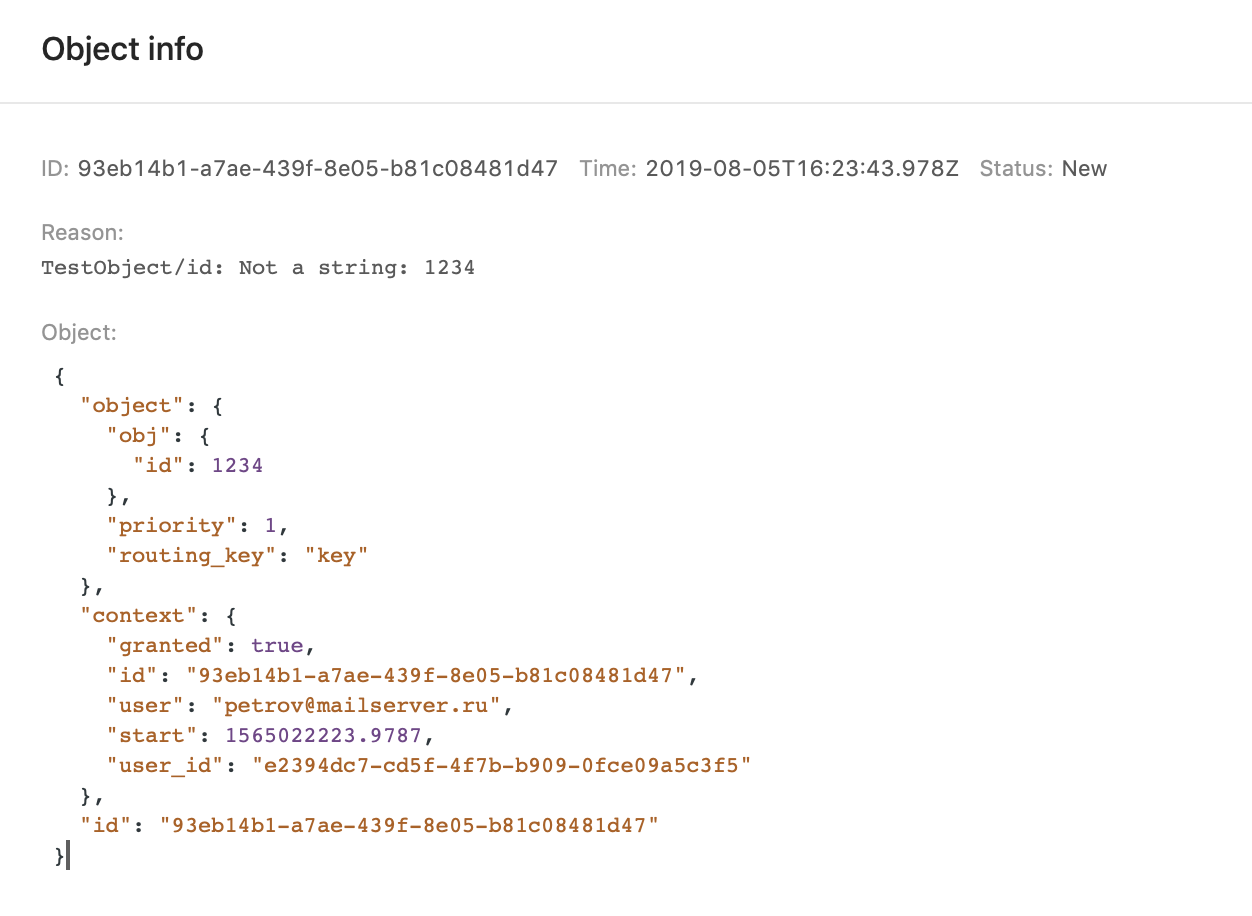
* Ошибка при обработке объекта в каком-либо из пайплайнов.
* Система TDG ожидает объект в определенном формате, но объект пришел из внешней системы в другом формате.
* Внутренняя ошибка системы.
* Сбой оборудования (hardware).

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.

### 4.1. 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(см руководство по эксплуатации). Настройки почтового сервера и подписчиков задаются через web-интерфейс на вкладках **Settings > Mail** server и **Settings > Subscribers** соответственно.

#### 4.1.1. 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.

#### 4.1.2. 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.

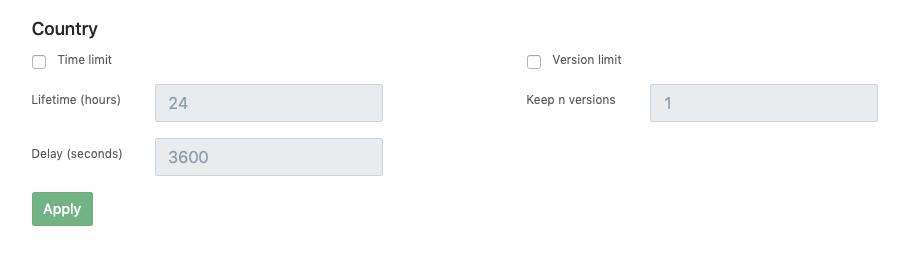
## 5. Управление бизнес-объектами

### 5.1. 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 работе 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

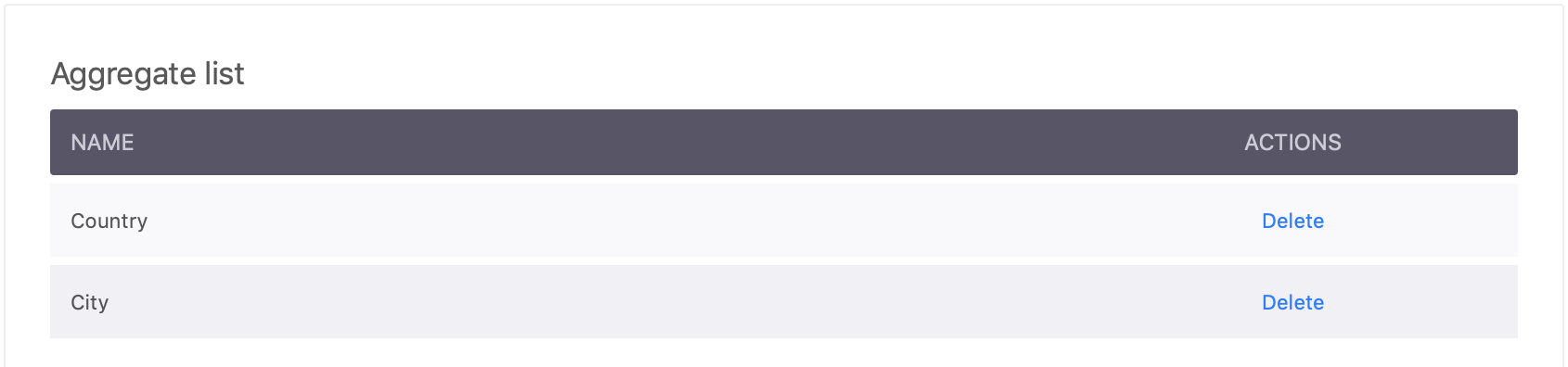
* Lifetime (hours) — время жизни объекта в часах. Значение по умолчанию: 24.
* Delay (seconds) — интервал в секундах, через который запускается очередная проверка устаревших объектов и их удаление. Значение по умолчанию: 36000.

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.

### 5.2. 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.

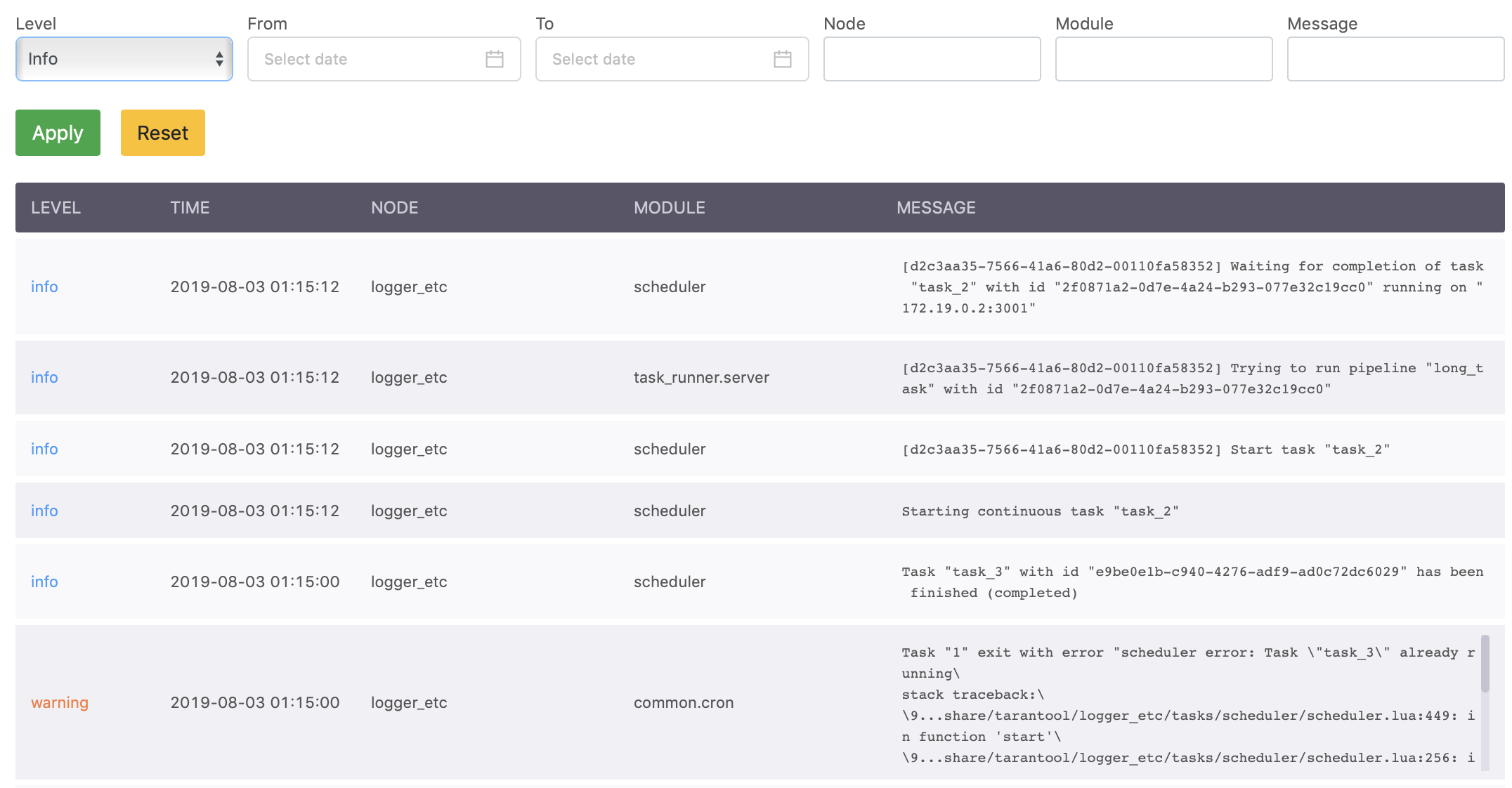


При удалении определенного типа также удаляются из storage все объекты указанного типа вместе с зависимостями (объекты логических типов «Entity» и «ValueObject»). При этом если объект типа «Entity» используется другим агрегатом, то он не удаляется. Подробнее про логические типы объектов см. раздел «Разработка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

## 6. 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](#section-logger).



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](#script-logs).

## 7. Репликация объектов

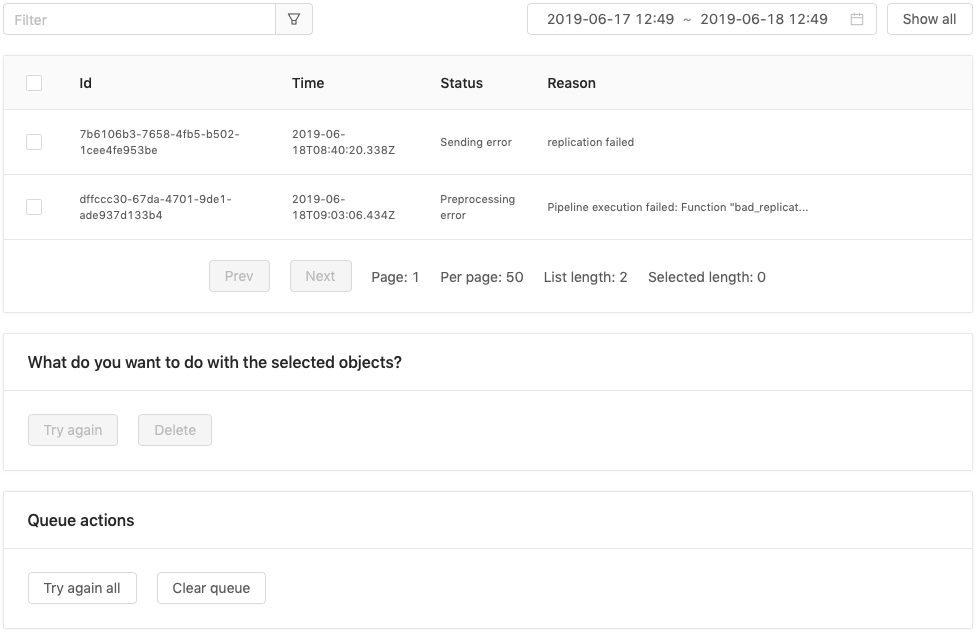
Механизм репликации объектов позволяет отправлять объекты во внешние системы в нужном формате.

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

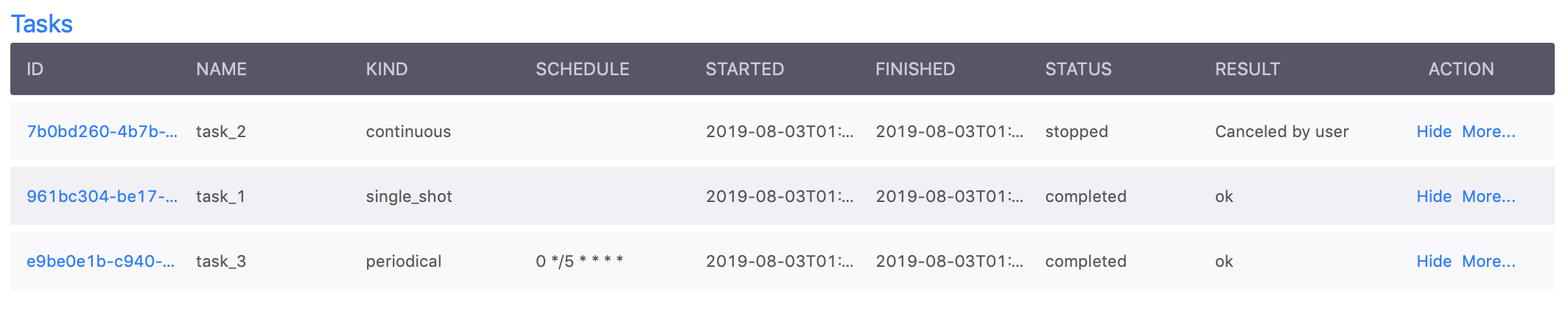
При повторной операции (**Try again**) над объектом его статус меняется на «In Progress». Если повторная операция успешна, объект переходит на следующий этап обработки или удаляется из ремонтной очереди (в зависимости от предыдущего статуса). Если повторная операция завершилась ошибкой, статус объекта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

## 8. 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).

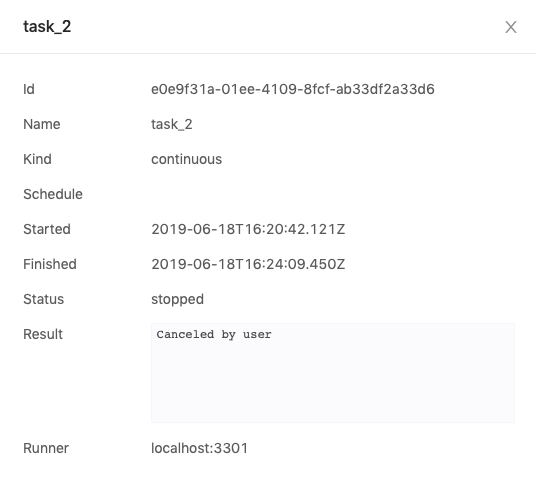
Отслеживать текущее состояние задач и управлять их выполнением можно через web-интерфейс на вкладке **Tasks**.



* **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**.



## 9. Тестирование и отладка

Web-интерфейс администратора предоставляет ряд инструментов для тестирования и отладки работоспособности системы. Они доступны на вкладках **Model**, **Test**, **Graphql** и **Console**.

### 9.1. Model

Дает возможность загрузить в систему модель данных или проверить и отредактировать уже существующую.



Если в систему уже была загружена конфигурация вместе с моделью данных, в окне **Request** будет показана текущая модель.

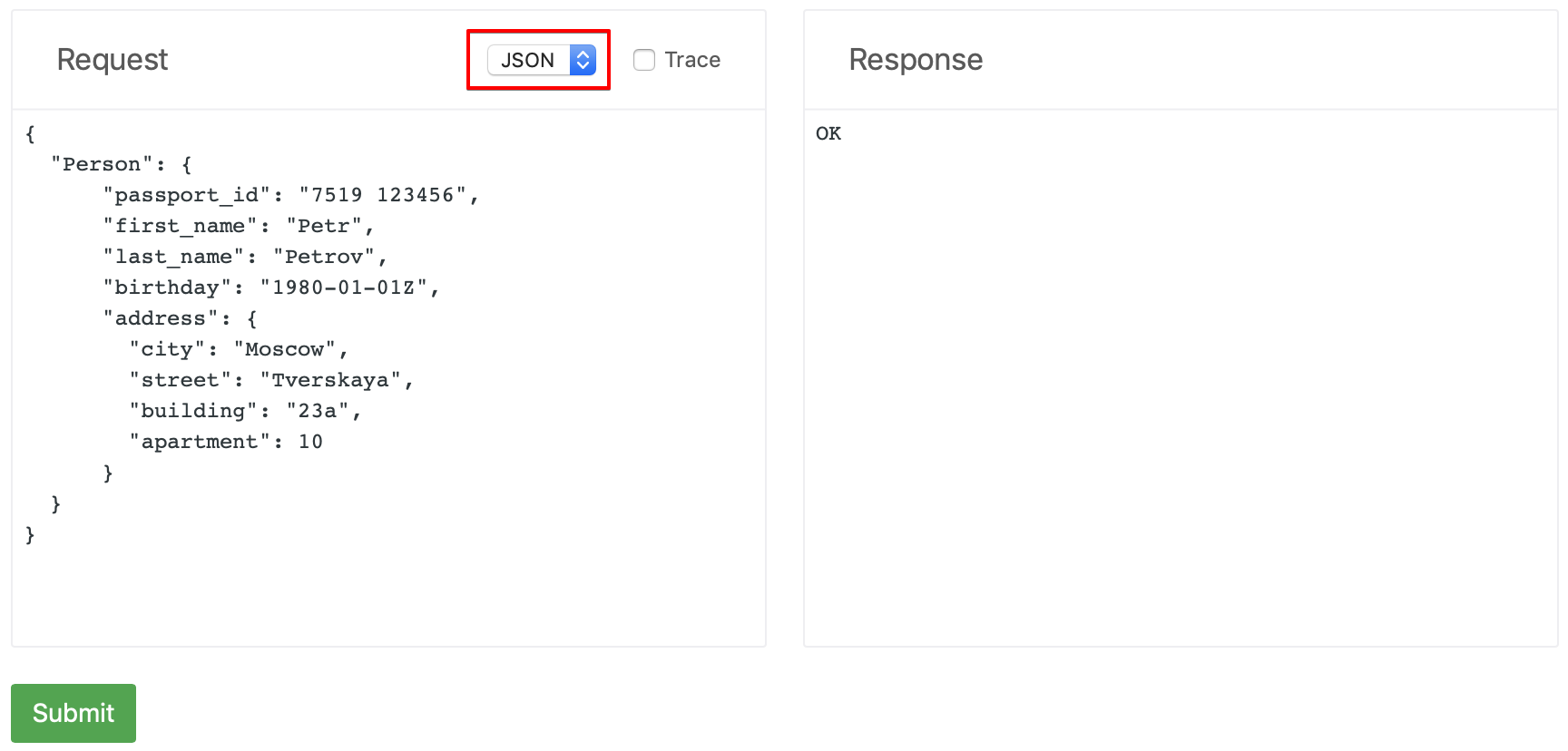
Если необходимо, можно внести правки в модель и снова загрузить ее в систему, нажав **Submit**. При успешной обработке и загрузке, в качестве ответа от системы в окне **Response** будет отображена загруженная модель. В случае ошибки в окне **Response** будет выведено сообщение об ошибке.

### 9.2. Test

На этой вкладке можно протестировать обработку системой входящего запроса.

Если система уже готова к работе (настроен кластер, определены роли инстансов, загружена конфигурация системы и модель данных), в окно **Request** можно вставить текст запроса и отправить его, нажав **Submit**.

Запросы можно отправлять в формате JSON или SOAP XML, выбрав нужный формат переключателем в окне **Request**.



В окне **Response** выводится ответ от соответствующего модуля системы, отвечающего за прием и первоначальную обработку (parsing) запроса. В случае ошибки синтаксиса запроса, выводится сообщение об ошибке. Если запрос корректен, выдается ответ «ОК».

Ответ «ОК» означает, что запрос успешно обработан на роли connector и отправлен дальше на обработку на роль input\_processor. Был ли запрос успешно обработан далее и сохранен в системе — нужно проверить дополнительно. Прежде всего нужно проверить [ремонтную очередь](#repair-queue): в случае ошибок обработки объект окажется там и не будет сохранен в систем. В этом случае можно проанализировать ошибки и внести коррективы в сам запрос и/или его обработчики. Если ремонтная очередь пустая, значит объект успешно обработан и сохранен. Проверить это можно, используя другие средства отладки, в частности [отправив GraphQL-запрос](#tab-graphql).

**Пример**

Ниже приведен пример описания модели данных и входящего запроса (объекта) для этой модели. Подробнее о модели данных для приложения TDG см. раздел «Разработка доменной модели» в руководстве по эксплуатации.

Модель:

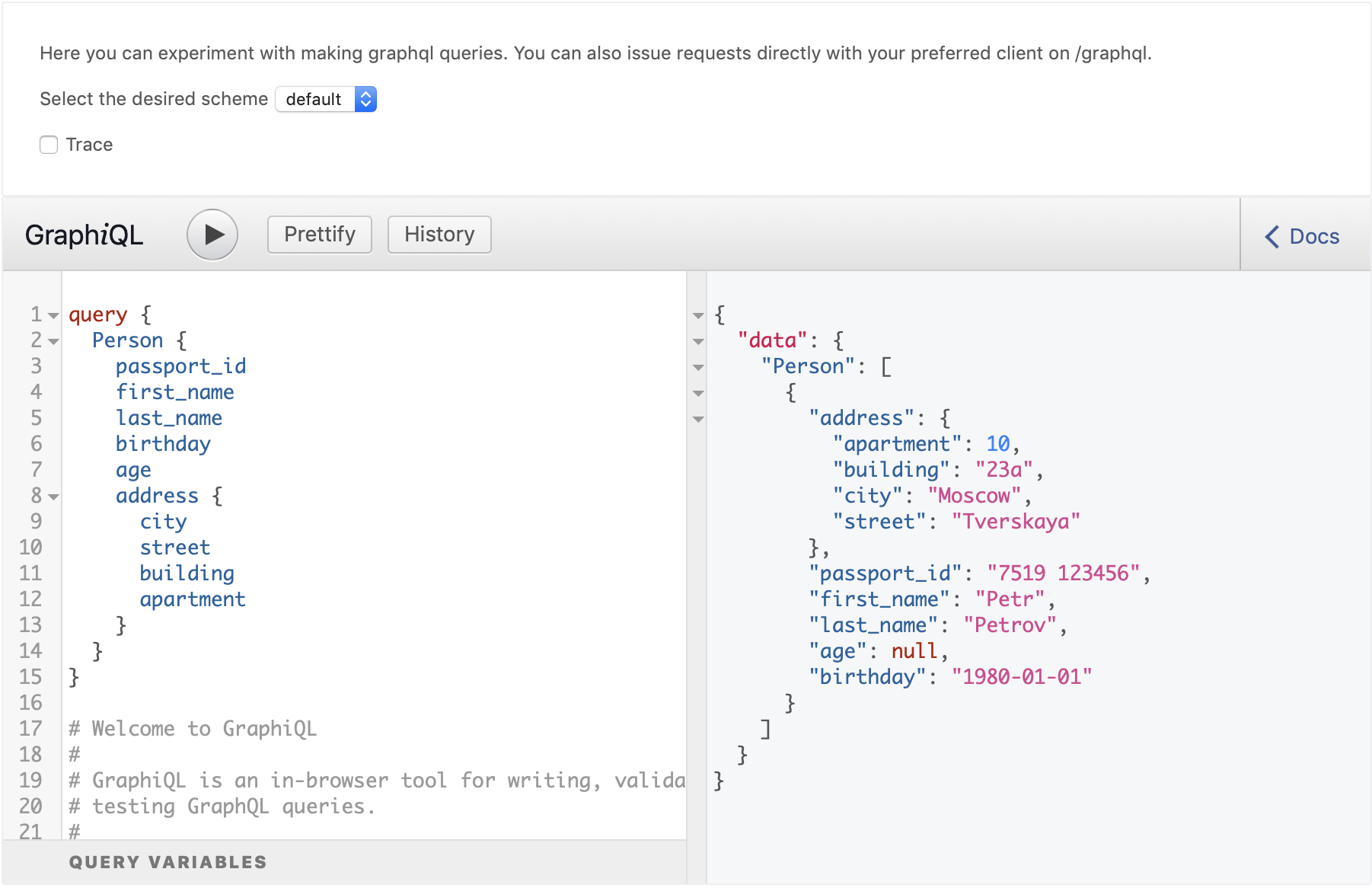
[  
 {  
 "name": "Address",  
 "type": "record",  
 "logicalType": "ValueObject",  
 "doc": "Адрес",  
 "fields": [  
 {"name": "city", "type": "string"},  
 {"name": "street", "type": "string"},  
 {"name": "building", "type": "string"},  
 {"name": "apartment", "type": "int"}  
 ]  
 },  
  
 {"type": "record",  
 "name": "Person",  
 "logicalType": "Aggregate",  
 "doc":"Человек",  
 "fields": [  
 {"name": "passport\_id", "type": "string"},  
 {"name": "first\_name", "type": "string"},  
 {"name": "last\_name", "type": "string"},  
 {"name": "address", "type": "Address"},  
 {"name": "birthday", "type": {"type": "string", "logicalType": "Date"}},  
 {"name": "age", "type":["null", "int"]}  
 ],  
  
 "indexes": ["passport\_id"]  
  
 }  
]

Объект в формате JSON для отправки входящего тестового запроса:

{  
 "Person": {  
 "passport\_id": "7519 123456",  
 "first\_name": "Petr",  
 "last\_name": "Petrov",  
 "birthday": "1980-01-01Z",  
 "address": {  
 "city": "Moscow",  
 "street": "Tverskaya",  
 "building": "23a",  
 "apartment": 10  
 }  
 }  
}

### 9.3. Graphql

На этой вкладке доступна встроенная [утилита GraphiQL](https://github.com/graphql/graphiql), с помощью которой можно писать, валидировать и отправлять в систему запросы (query) и мутации (mutation) на языке запросов GraphQL.



Для работы с данными нужно использовать схему default (переключатель **Select the desired scheme** вверху страницы). Эта схема задана по умолчанию и используется для доступа к объектам.

**Пример**

Рассмотрим GraphQL запросы и мутации для примера модели данных и тестового объекта, которые обсуждались ранее в разделе [«Тестирование и отладка > Test»](#test-request-example). В предыдущем примере мы остановились на том, что в систему загружена модель данных и отправлен входящий запрос в формате JSON, в результате которого объект был корректно обработан и сохранен на роли storage.

1. Запрос на получение объекта:

query {  
 Person {  
 passport\_id  
 first\_name  
 last\_name  
 birthday  
 address {  
 city  
 street  
 building  
 apartment  
 }  
 }  
}

1. Мутация на вставку/изменение объекта и получение вставленного объекта:

mutation {  
 Person(insert:{passport\_id:"7519 123456"  
 first\_name:"Petr"  
 last\_name:"Petrov"  
 birthday:"1980-01-01Z"  
 address:{  
 city:"Moscow"  
 street:"Tverskaya"  
 building:"23a"  
 apartment:10  
 }  
 }  
 )  
 {  
 passport\_id  
 first\_name  
 last\_name  
 birthday  
 address {  
 city  
 street  
 building  
 apartment  
 }  
 }  
}

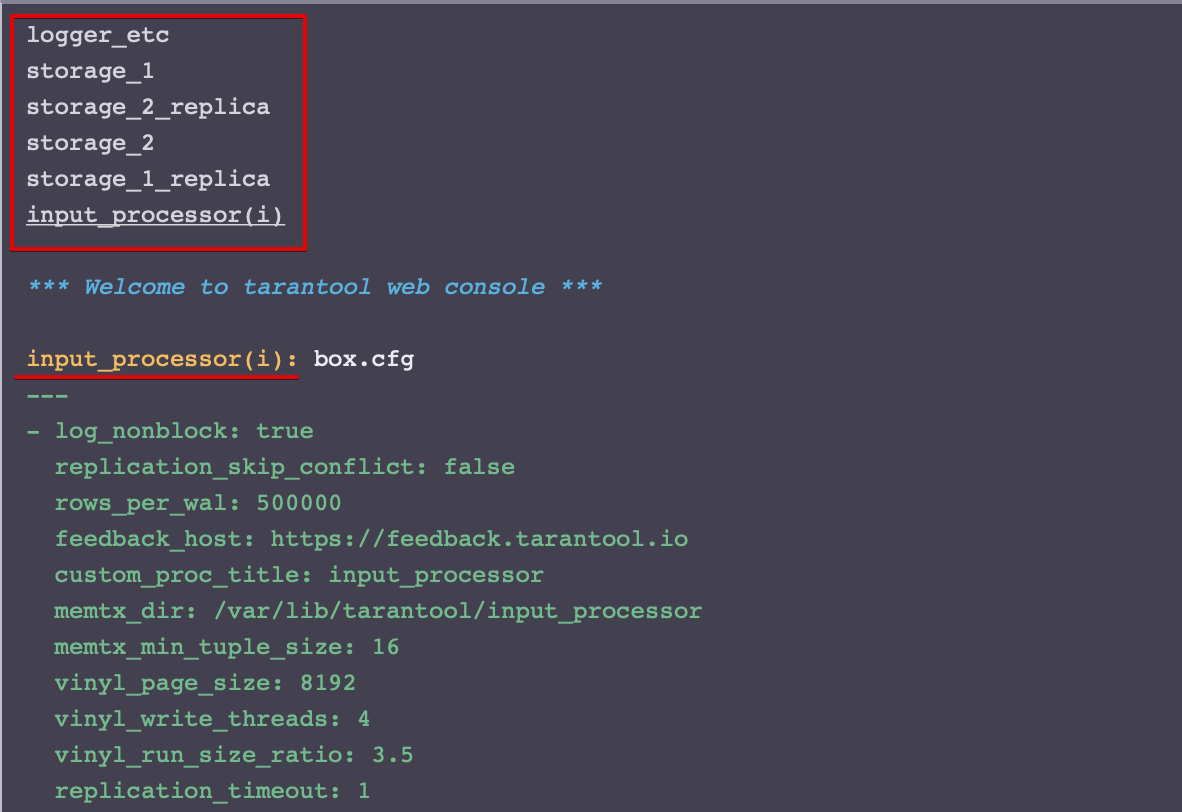
1. Мутация на удаление объекта и получение удаленного объекта:

mutation {  
 Person(passport\_id:"7519 123456" delete:true){  
 passport\_id  
 first\_name  
 last\_name  
 birthday  
 address {  
 city  
 street  
 building  
 apartment  
 }  
 }  
}

### 9.4. Console

На этой вкладке находится консоль для доступа к инстансам Tarantool, которая позволяет выполнять все действия, описанные в [основной документации по Tarantool](https://www.tarantool.io/ru/doc/2.1/).

В консоли доступно быстрое переключение между инстансами.



## 10. Метрики

Для мониторинга системы предоставляются метрики в формате [Prometheus](https://prometheus.io). Значения метрик доступны по адресу http://<IP\_адрес\_инстанса>/metrics для каждого из инстансов кластера. Для систем-сборщиков метрик необходимо подать на вход IP-адреса всех инстансов кластера.

Доступны следующие метрики типа [gauge](https://prometheus.io/docs/concepts/metric_types/#gauge):

* tarantool\_space\_tuples\_total — Общее число кортежей в спейсе. Выводятся значения для всех спейсов инстанса.
* tarantool\_memory\_limit\_bytes — Максимальный объем памяти, который может использовать Tarantool (в байтах).
* tarantool\_requests\_total — Общее число запросов (по типам запросов). Типы запросов:
  + auth
  + delete
  + update
  + select
  + error
  + replace
  + upsert
  + eval
  + insert
* tarantool\_memory\_used\_bytes — Объем памяти, используемый Tarantool’ом на данный момент (в байтах).
* tarantool\_tuples\_memory\_bytes — Объем памяти, выделенный под кортежи Tarantool’a (в байтах).
* tarantool\_uptime\_seconds — Время с момента старта данного инстанса (в секундах).
* tarantool\_system\_memory\_bytes — Объем памяти, используемый системой и индексами Tarantool’а (в байтах).

**Пример**

# HELP tarantool\_space\_tuples\_total Total number of tuples in a space  
# TYPE tarantool\_space\_tuples\_total gauge  
tarantool\_space\_tuples\_total{space\_name="log"} 3869  
tarantool\_space\_tuples\_total{space\_name="task\_list"} 0  
tarantool\_space\_tuples\_total{space\_name="scheduler\_task\_list"} 20  
  
  
# HELP tarantool\_memory\_limit\_bytes Maximum amount of memory Tarantool can use  
# TYPE tarantool\_memory\_limit\_bytes gauge  
tarantool\_memory\_limit\_bytes 134217728  
  
# HELP tarantool\_requests\_total Total number of requests by request type  
# TYPE tarantool\_requests\_total gauge  
tarantool\_requests\_total{request\_type="call"} 385  
tarantool\_requests\_total{request\_type="auth"} 1  
tarantool\_requests\_total{request\_type="delete"} 162  
tarantool\_requests\_total{request\_type="update"} 327  
tarantool\_requests\_total{request\_type="select"} 414  
tarantool\_requests\_total{request\_type="error"} 0  
tarantool\_requests\_total{request\_type="replace"} 0  
tarantool\_requests\_total{request\_type="upsert"} 0  
tarantool\_requests\_total{request\_type="eval"} 0  
tarantool\_requests\_total{request\_type="insert"} 1219  
  
# HELP tarantool\_memory\_used\_bytes Amount of memory currently used by Tarantool  
# TYPE tarantool\_memory\_used\_bytes gauge  
tarantool\_memory\_used\_bytes 33554432  
  
# HELP tarantool\_tuples\_memory\_bytes Amount of memory allocated for Tarantool tuples  
# TYPE tarantool\_tuples\_memory\_bytes gauge  
tarantool\_tuples\_memory\_bytes 2461864  
  
  
# HELP tarantool\_uptime\_seconds Number of seconds since the server started  
# TYPE tarantool\_uptime\_seconds gauge  
tarantool\_uptime\_seconds 36415  
  
# HELP tarantool\_system\_memory\_bytes Amount of memory used by Tarantool indexes and system  
# TYPE tarantool\_system\_memory\_bytes gauge  
tarantool\_system\_memory\_bytes 31092568

## 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 (см руководство по эксплуатации) 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» (см руководство по эксплуатации). 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](#section-tconnect) — конфигурация роли connector. Настраиваются:
  + *input*
  + *output*
  + *routing*
* [input\_processor](#section-input-processor) — конфигурация роли input\_processor. Настраиваются:
  + *classifiers*
  + *routing*
  + *storage*
* [output\_processor](#section-output-processor) — настройки для роли output\_processor.
* [repair\_queue](#section-repair-queue) — настройки обработки объекта при попадании в ремонтную очередь.
* [logger](#section-logger) — 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](#section-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.

#### 14.1.4. 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. Устранение неисправностей

См. инструкции в документации на сопряженное ПО:

* Tarantool версия для предприятий:

<https://www.tarantool.io/en/enterprise_doc/1.10/admin/#monitoring-cluster-via-cli>

* Tarantool версия с открытым кодом:

<https://www.tarantool.io/en/doc/1.10/book/admin/troubleshoot/>

## 13. Восстановление после сбоя

См. инструкции в документации на Tarantool с открытым кодом:

<https://www.tarantool.io/en/doc/1.10/book/admin/troubleshoot/>

## 14. Резервное копирование

См. инструкции в документации на Tarantool с открытым кодом:

<https://www.tarantool.io/en/doc/1.10/book/admin/backups/>

## 15. Приложение

### 15.1. 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 в формате JSON.

#### 15.1.1. 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.

#### 15.1.2. 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"]  
 },

#### 15.1.3. 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.

#### 15.1.4. 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).

#### 15.1.5. 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.
* config.backup.yml — резервная копия файла конфигурации системы.

#### 15.1.6. restore

Используется для восстановления данных из резервных копий инстансов.

Формат:

tdgctl.py restore

#### 15.1.7. stop

Используется для остановки всех инстансов.

Формат:

tdgctl.py stop

#### 15.1.8. start

Используется для старта всех инстансов.

Формат:

tdgctl.py start