# МПК Пересвет. Архитектура

## Описание

Платформа МПК Пересвет – надстройка над базой данных реального времени, позволяющая строить модели технических объектов и выполнять на этих моделях вычисления. Также в рамках модели описывается информационная система в целом.

В качестве хранилища данных реального времени может выступать большой спектр баз данных: PostgreSQL, Victoriametrics и любая другая, в случае, если для неё создан драйвер. Причём разные хранилища могут использоваться одновременно.

## Модель

Модель технического объекта делится на статическую и динамическую.

Статическая модель – иерархия. Иерархия строится на базе сервера OpenLDAP, что позволяет не только строить иерархии любой сложности, но и выполнять аутентификацию/авторизацию пользователей.

Динамическая модель – совокупность методов, выполняемых в момент возникновения того или иного события.

## События

События, которые могут возникать в модели и, соответственно, запускать расчёты: изменение тега, возникновение тревоги, событие по расписанию. В дальнейшем, возможно, добавятся события пропадания и квитирования тревоги.

## Сущности модели

Сущности, на базе которых строится модель технического объекта:

1. Объект;
2. Тег;
3. Тревога;
4. Метод;
5. Константа;
6. Расписание.

Сущности, участвующие в описании модели информационной системы:

1. Хранилище данных;
2. Коннектор.

## Микросервисы

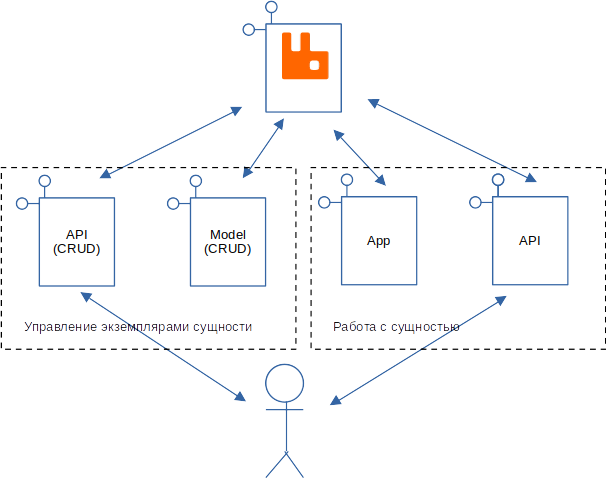
Платформа МПК Пересвет имеет микросервисную архитектуру.

Для каждой сущности модели создаётся набор сервисов, каждый из которых работает как самостоятельное приложение.

Сервисы между собой общаются с помощью брокера сообщений RabbitMQ.

В общем случае для каждой сущности создаётся четыре сервиса:

1. API (CRUD).  
   Сервис точек входа API. Главная задача – валидация средствами Pydantic входных запросов.
2. Model (CRUD).  
   Сервис, выполняющий работу команд Create-Read-Update-Delete.
3. App.  
   Сервис, выполняющий основную функциональность сущности (совокупность подобных сервисов – ядро платформы).
4. API  
   Сервис точек входа для API сущности, если она предоставляет какую-то функциональность внешним клиентам.



Разделение сервисов API(CRUD) и Model(CRUD) позволяет гибче управлять версионностью API. Отделение сервисов API(CRUD) и Model(CRUD) от App и API позволяет выделить работу по управлению экземплярами сущностей вообще на отдельный компьютер, тем более, что эта работа выполняется, обычно, только при создании модели технического объекта и очень редко в дальнейшем.

Дробление сервисов на такие мелкие части не является проблемой, так как сервер приложений Nginx.unit позволяет запускать внутри одного своего экземпляра сразу несколько приложений, таким образом, в рамках одного контейнера можно запустить сразу несколько микросервисов.

Рассмотрим разделение функциональности между сервисами на примере сущности «alerts»:

1. API (CRUD)   
   Команды, с помощью которых пользователь создаёт экземпляры тревог, привязывает их к тегам, описывает их параметры и т.д., то есть расширяет тревогами иерархическую модель объекта.  
   Сервис выполняет две задачи: проверка корректности входных данных запроса от пользователя и создание сообщения для сервиса «Model (CRUD)».
2. Model (CRUD)  
   Сервис также выполняет две задачи: выполняет ту функциональность, которую клиент запрашивает через «API (CRUD)» и создаёт сообщение для сервиса «App». Впрочем, на эти сообщения могут быть подписаны и другие сервисы.
3. App  
   Сервис управляет процессом возникновения/пропадания тревог, будучи подписан на события изменения тегов. Также сервис порождает сообщения о возникновении/пропадании/квитировании тревог.
4. API  
   Сервис точек входа для доступа к функциональности сервиса «App». С помощью этого сервиса клиент может получить список активных тревог (другой способ – подписаться на сообщения), а также может квитировать тревогу.

Не обязательно для каждой сущности создавать все 4 сервиса.

К примеру, сущность «const» будет иметь только два сервиса: API(CRUD) и Model(CRUD), так как никакой функциональности для констант не предполагается – они будут использоваться только как параметры для передачи в методы.

## Правила именования главных файлов сервисов

API(CRUD): <сущность>\_api\_crud\_svc.py

Model(CRUD): <сущность>\_model\_crud\_svc.py

App: <сущность>\_app\_svc.py

API: <сущность>\_api\_svc.py

Файлы с конфигурациями для каждого сервиса имеют в именах вместо svc слово settings: <сущность>\_api\_crud\_settings.py.

Базовые классы, от которых наследуются сервисы, содержатся в файлах без имени сущности: api\_crud\_svc.py (см. также структуру проекта ниже).

## Компоновка

Как упоминалось выше, Nginx.unit позволяет запускать внутри себя одновременно несколько приложений.

Соответственно, в самом простом случае разворачивания платформы набор контейнеров может быть такой:

1. Ядро платформы и все API (совокупность сервисов «API(CRUD)», «Model(CRUD)», «App», «API»).
2. OpenLDAP
3. PostgreSQL
4. RabbitMQ
5. Celery
6. Все вычислительные методы, то есть динамическая модель.

В других случаях возможны совершенно любые комбинации разбиения контейнеров 1 и 6, плюс запуск конейнера с Nginx в качестве прокси.

## Преимущества

Помимо очевидных плюсов в плане масштабируемости и возможностей высокой доступности существуют другие практические выгоды от подобного решения.

Допустим, потенциальному заказчику нужно установить демоверсию платформы на какой-то срок. Соответственно, мы создаём модель какого-то технического объекта у заказчика, а затем, на срок действия демоверсии, просто убираем сервисы «API (CRUD)» и «Model (CRUD)» для всех сущностей. Таким образом, отсутсвует возможность расширения модели объекта.

## Недостатки

1. При небрежном планировании сервисов можно создать бардак
2. Скорость межпроцессного взаимодействия. Особенно влияние этого взаимодействия на выполнение команд «data/set» и «data/get».

## Структура каталогов

Ниже представлена структура каталогов проекта

.

├── certificates

│ # скрипты для генерации сертификатов, а также сами сертификаты

│ ├── 01. gen\_root.sh

│ ├── 02. gen\_server.sh

│ ├── 03. gen\_client.sh

│ └── gen\_crt.sh

├── config

│ # конфигурации внешних компонентов системы, то есть не нашей разработки

│ # как показано на примере nginx, скорее всего, для каждого компонента будут две папки:

│ # безопасная конфигурация и без учёта безопасности (TLS и без TLS)

│ ├── celery

│ ├── nginx

│ │ ├── no-security

│ │ └── security

│ ├── postgresql

│ ├── rabbitmq

│ └── victoriametrics

├── docker

│ ├── compose

│ │ # docker-compose файлы для компонентов системы, для всей системы в целом

│ └── docker-files

│ # docker файлы для сборки компонентов системы

│ └── ldap

│ # файлы для создания ldap-контейнера

├── docs

│ # документация на систему, на стандартные компоненты.   
│ # Как решать вопрос с дополнительными компонентами — пока не понятно.

├── src

│ # исходный код системы   
│ # каталог, исходные тексты из которого формируют docker-контейнеры

│ ├── common

│ │ # общие модули, используемые всеми сервисами (логирование, работа с ldap,

│ │ # базовые классы, от которые наследуются сервисы и т.д.

│ │ ├── \_\_init\_\_.py

│ │ │ # журнал, используемый всеми сервисами

│ │ ├── logger.py

│ │ │ # класс для работы с иерархией, используемый всеми сервисами

│ │ ├── hierarchy.py

│ │ │ # базовый класс сервиса в иерархии сервисов

│ │ ├── svc.py

│ │ │ # базовый класс настроек, от которого наследуются настройки для каждого сервиса

│ │ ├── settings.py

│ │ │ # файл с классами, наследуемыми от класса BaseModel модуля pydantic

│ │ │ # для проверки корректности входных данных команд CRUD

│ │ ├── api\_crud\_svc.py

│ │ │ # базовый класс для всех сервисов <сущность>\_model\_crud\_svc.py

│ │ ├── model\_crud\_svc.py

│ │ │ # базовый класс для всех конфигураций сервисов <сущность>\_model\_crud\_svc.py

│ │ └── model\_crud\_settings.py

│ └── services

│ # каждый каталог в этой папке — строго отдельный сервис,

│ # исходные коды которого пишем мы

│ # каждый сервис, в свою очередь,

│ # сам разбивается на несколько микросервисов

│ # ниже — пример на базе сервиса работы с тревогами

│ └── alerts

│ # все исходники, связанные с сервисом работы с тревогами

│ ├── api

│ # микросервис API, которое предоставляет клиенту функциональность работы

│ # с тревогами: получение списка активных тревог, квитирование тревоги и т.д.

│ │ └── api\_v1

│ │ # версии API

│ ├── api\_crud

│ │ # микросервис API, предоставляющий функциональность

│ │ # создания-чтения-обновления-удаления тревог, то есть всего того, что касается

│ │ # иерархической модели объекта

│ │ ├── api\_v1

│ │ # версии API

│ ├── app

│ # микросервис, реализующий логику работы с тревогами   
 # (исключая создание-обновление-удаление и т.д.)

│ └── model\_crud

│ │ # микросервис, реализующий логику создания-обновления-удаления-и т. д. Тревог   
│ │ # почему этот сервис отделён от api-crud — API обновляется чаще,   
│ │ # чем логика работы с сущностями;   
│ │ # кроме того, добавляется возможность поддерживать одновременно   
│ │ # несколько версий API

└── tests

│ # каталог с тестами

│ ├── load

│ │ # нагрузочные тесты

│ └── unit

│ │ # юнит-тесты

## Обмен сообщениями

В общем случае сервис, стартуя, подписывается на очередь, из которой забирает сообщения для обработки. Если запускаются несколько экземпляров сервиса, то они подписываются на одну очередь и забирают из неё сообщения в конкурентном режиме.

Кроме этого, сервис создает exchange типа direct, куда публикует свои события.

Разберём на примере сервисов, работающих с тегами.

### tags\_api\_crud

Задача этого сервиса, как и всех сервисов «\*\_api\_crud» - проверка корректности пришедших от клиента данных, которая выполняется с помощью модуля Pydantic.

В конфигурации сервиса указывается имя exchange’а типа direct, куда сервис публикует задачи CRUD с уже проверенными исходными данными.

Правило по умолчанию: каждый сервис создает exchange типа direct co своим именем, куда публикует свои события. То есть сервис tags\_api\_crud создаёт exchange с именем tags\_api\_crud.

Для реализации команды create сервис использует RPC к сервису tags\_model\_crud, так как ожидает id вновь созданного тега.

### tags\_model\_crud

Сервис выполняет логику работы с тегами в иерархической модели, то есть задачи создания/обновления…: всё, что касается изменения узлов в иерархической модели.

В конфигурации сервиса указывается имя exchange’а, на который ему нужно подписаться, а также имя очереди, из которой все экземпляры этого сервиса будут забирать сообщения. То есть при старте сервис подписывается на exchange с именем tags\_api\_crud. Очередь, допустим, tags\_model\_crud. Таким образом, все экземпляры сервиса будут забирать сообщения из одной очереди.

Также сервис создаёт exchange tags\_model\_crud типа direct, в который публикует сообщения об изменениях в тегах. Все другие сервисы, заинтересованные в получении сообщений об изменениях в тегах должны подписываться именно на этот exchange, а не на tags\_api\_crud.

### tags\_app

Сервис выполняет работу с самими тегами. В настоящий момент — отсечение лишних значений (пока даже и этого нет).

Сервис может быть подписан на exchange tags\_model\_crud, чтобы оперативно получать данные об изменениях тегов. Это может быть актуально в том случае, если сервис делает кэш параметров тегов. Если же каждый раз для своей работы обращается к иерархии, то смысла в этой подписке нет.

Сервис подписывается на exchange data\_set.