Фреймворк для AI/ML-проектов

(R&D-версия)

Руководство администратора

Skoltech

Веб-страница проекта: http://dataskai.com

Содержание

1 Установка и настройка

В руководстве указаны основные положения при развертывании версии DATASKAI с использованием сети ”Интернет”. Актуальные уточнения и дополнения опубликованы на официальном сайте фреймворка.

Для установки версии R/D в изолированной среде обратитесь к файлу помощи instruction.txt, расположенном после распаковки архива.

1.1 Подсистемы

В руководстве рассмотрен процесс установки и настройки следующих подсистем:

- Data Ingestion Subsystem
- Data Storage Subsystem
- Data Processing Subsystem
- Data Analysis Subsystem

1.1.1 Data Ingestion Subsystem

Установка подсистемы включает установку следующих внешних зависимостей и компонентов.

Зависимости:

- https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.0/index.html
- https://kafka.apache.org/
- https://docs.confluent.io/current/kafka-rest/index.html

Компоненты:

- Source connectors
- Data processors
- Sink connectors

Установка внешних зависимостей

Data Bus (шина данных)

Установите Apache Kafka версии 2.0.0 или выше.

Рекомендуемый порядок установки:

- Установите Hortonworks Data Platform.
- Установите и настройте Apache Kafka согласно документации https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.0/installing-configuring-kafka
  /content/installing_kafka.html.

Data Gateway (шлюз данных)

Установите Confluent REST Proxy, следуя https://docs.confluent.io/current/kafka-rest/index.htmlinstallation.

Инструкции по установке

Source Connectors

Source connectors устанавливаются отдельно согласно документации на конкретный коннектор.

Data Processors (процессоры данных)

Процессоры данных устанавливаются отдельно в соответствии с документацией.

Sink Connectors

Sink connectors устанавливаются отдельно в соответствии с документацией.

1.1.2 Data Storage Subsystem

Установка подсистемы включает установку следующих внешних зависимостей и компонентов.

Зависимости:

- https://www.postgresql.org/docs/
- https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.0/index.html

Компоненты:

- http://iot-serv2.skoltech.ru:85/en/components/data_service/component_install.htmldata-service-install-ref

Установка внешних зависимостей

Набор используемых хранилищ данных может отличаться в зависимости от домена приложения. Минимальная конфигурация требует установки PostgreSQL.

PostgreSQL

Поддерживается PostgreSQL 11 или. Обратитесь к https://www.postgresql.org/docs/.

HDFS

Поддерживается HDFS 3.0 или выше. Рекомендуемая процедура установки – развернуть HDFS с помощью https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.0/index.htmlHortonworks Data Platform.

Инструкции по установке Data Service

Приложение работает в контейнерной среде, поэтому базовым предварительным условием является настроенный докер.

Образ службы основан на следующих пакетах докеров:

— ubuntu: 18.04 и выше; — openjdk: 11 и выше; — python: 3.6 и выше.

Сборка / установка

Шаги сборки / установки:

1) сервис должен быть построен с использованием Spring Framework. Применяются автоматические установки с использованием файла конфигурации, поэтому сначала вы должны отредактировать файл конфигурации application-compose_api.yml (см. раздел конфигурации ниже);

2) перейдите в корневой каталог platform-proto-java. Создайте файл .jar с помощью gradle:

```
./gradlew :data-service-impl:clean && ./gradlew
 ↪  :data-service-impl:bootJar
```

3) создайте каталог для журналов (в этом примере workdir / logs), создайте образ докера и запустите контейнер.

```
mkdir -p workdir/logs

cd data-service-impl

docker build -t idcp/data-service-api .

docker run -d \
--name data-service-api \
--mount type=bind,source=${PWD}/../workdir/logs,target=/logs \
-p 8170:8170 \
-e SPRING_PROFILES_ACTIVE='compose_api' \
-v ${PWD}/application-compose_api.yml:/application-compose_api.y
 ↪  ml
 ↪  \
idcp/data-service-api
```

Также вы можете использовать docker-compose:

```
docker-compose -f docker-compose.<profile>.yml build
 ↪  data-service-api

docker-compose -f docker-compose.<profile>.yml up -d
 ↪  data-service-api
```

4) проверьте статус контейнера:

```
docker ps --filter "name=data-service-api"
```

STATUS Up означает, что контейнер успешно установлен и запущен.

Описание файла конфигурации

Сервис использует файл конфигурации Spring Boot application.yml.

```
spring:
  profiles:
    active: development
  datasource:
    driver-class-name: org.postgresql.Driver
    url:
     ↪  jdbc:postgresql://localhost:5432/${dataservice.namespace}
    username: <username>
    password: <password>
    hikari:
      minimum-idle: 1
      maximum-pool-size: 4
dataservice:
  data-catalog: DataCatalogPSQLImpl
  data-query: PSQLDataQueryImpl
  namespace: <namespace>
  schema: public
  proxy: psql
server:
  port : 8170
logging:
  level:
    platform: DEBUG
    org.springframework: INFO
    org.hibernate: ERROR
```

Файл конфигурации состоит из из следующих строк.

SPRING PROPERTIES

PROFILES – позволяет определить, какие профили используют эту конфигурацию.

spring.profiles.active - -описывает свойство, определяющее, какие профили активны (строка, например, «разработка»).

Подключения к рабочей базе данных также можно настроить автоматически с помощью пула DATASOURCE.

spring.datasource.driver-class-name – полное имя драйвера JDBC (строка, например, «org.postgresql.Driver»).

spring.datasource.url – URL-адрес JDBC базы данных ‘jdbc: postgresql: // <hostname>: <port> / $ {dataservice.namespace}‘ (url, например, ‘jdbc: postgresql: // localhost: 5432 / $ { dataservice.namespace} ‘).

spring.datasource.username – логин пользователя базы данных (строка, например, «имя пользователя»).

spring.datasource.password – пароль для входа в базу данных (например, «пароль»).

spring.datasource.hikari. * – специальные настройки для hikari.

КОНФИГУРАЦИЯ ДАТАСЕРВИСА

dataservice.data-catalog -DataCatalogPSQLImpl - имя компонента службы каталога данных.

dataservice.data-query -PSQLDataQueryImpl – имя компонента службы запроса данных.

dataservice.namespace – имя базы данных (строка, например, «test-data-api»).

dataservice.schema – имя схемы (строка, например, «общедоступная»).

dataservice.proxy – имя прокси (строка, например, «psql»).

ВСТРОЕННАЯ КОНФИГУРАЦИЯ СЕРВЕРА

server.port – HTTP-порт сервера (int, например 8190).

СВОЙСТВА ЛОГОВ

logging.level. * – сопоставление уровней логов. Например, ‘logging.level.org.springframework = INFO‘.

Запуск службы

Чтобы запустить службу, выполните команду docker:

```
docker start data-service-api
```

Также возможно использование docker-compose:

```
docker-compose up -d data-service-api
```

После запуска вы можете отправить запрос в сервис, например:

```
[
    {"namespace":"aerophm","name":"aids_reports_01","versions":[
    ↪  "1.0"]},
    {"namespace":"aerophm","name":"aids_reports_02","versions":[
    ↪  "1.0"]},
    {"namespace":"aerophm","name":"aircrafts","versions":["1.0"]}
]
```

Останов сервиса

Чтобы остановить сервис, используйте интерфейс командной строки докера:

```
docker stop data-service-api
```

или используйте docker-compose:

```
docker-compose down data-service-api
```

Удаление сервиса

Чтобы удалить службу, используйте команду docker:

```
docker rm data-service-api
```

или используйте docker-compose:

```
docker-compose rm data-service-api
```

1.1.3 Data Processing Subsystem

Процесс установки рассмотрен в онлайн-версии документации.

1.1.4 Data Analysis Subsystem

Процесс установки рассмотрен в онлайн-версии документации.

1.2 Компоненты

В руководстве рассмотрен процесс установки следующих компонентов:

- Evaluation tools
- Toolkit CLI
- Submits Web App
- Metrics Service
- Project Wizard
- Prediction Builder
- Feature Store
- Models Player
- Model Wrapper
- Feature Builder
- Data Service
- Subject Domain Cache Service

1.2.1 Evaluation tools

Процесс установки рассмотрен в онлайн-версии документации.

1.2.2 Toolkit CLI

Инструменты для облегчения операций обработки данных DATASKAI.

Установка

Все инструменты работают напрямую из командной строки без какой-либо установки, но требуется проверить требования pip из ./requirements.txt

Установка требований:

```
pip install -r ./requirements.txt
```

pip install -r ./requirements.txt

1.2.3 Submits Web App

Отображает графический интерфейс таблицы лидеров для сабмитов.

Необходимые условия

Сервис работает в контейнерной среде, поэтому базовым условием является настроенный Docker.

Сборка / установка

Проект содержит скрипт scripts/build_docker_image.sh для построения docker-образа сервиса. Запустите его для создания образа докера:

```
./scripts/build_docker_image.sh -t submits_web_app:1.0
```

Параметры контейнера:

- -t – тег, который будет использоваться для запуска контейнера (необязательный).

Описание файла конфигурации

Файл конфигурации написан в формате json и находится в каталоге configs. Всего существует 3 файла конфигурации: production.json, development.json, test.json, для работы в соответствующих средах обслуживания. Вы можете редактировать каждый из них.

Пример структуры файла конфигурации:

```
{
  "logging": {
    "filename": "./logs/development.log",
    "level": "INFO"
  },
  "board_templates": {
    "template_1": {
      "db": {
        "host": "localhost:27017",
        "database": "aero_db",
        "collection": "submits"
      },
```

```
      "config_db": {
        "host": "localhost:27017",
        "database": "aero_db",
        "collection": "submitter_configs"
      },
      "metrics": [
        {"name": "roc", "precision": 4}
      ],
      "frontend": {
        "text_fields": [
          "model_name"
        ],
        "sort_columns": [
          {
            "field": "roc",
            "ascending": false
          }
        ],
        "searchable_columns": [
          "model_name"
        ],
        "download_available": true,
        "fields": ["id"]
      }
    }
  },
  "boards": {
    "board_1": {
      "board_template": "template_1",
      "config_db": {"config_name": "aero__fw__eng_2_fuel_filter_
      ↪  clog__w_21_classification_v1"}
    },
    "board_2": {
      "board_template": "template_1",
      "config_db": {"config_name": "aero__fw__ENG_2_REV_PRESSURI
      ↪  ZED__w_21_classification_v1"}
    }
  }
}
```

Файл конфигурации состоит из следующих разделов.

— logging – необязательно, настройки ведения журнала.

– filename - путь к лог-файлу;

– level – необязательный, уровень ведения журнала. По умолчанию: INFO.

— board_templates – необязательные, шаблоны, которые можно использовать для расширения конфигов плат, описанных в разделе «доски». Это словарь <имя шаблона>: <шаблон>.

— boards – отдельные конфигурации платы, которые будут отображаться в графическом интерфейсе веб-приложения Submit. Помимо полей базовой конфигурации (см. Список ниже) каждая плата может содержать дополнительное поле board_template для использования конфигурации из шаблона.

Конфигурация борда (или шаблона борда) содержит следующие поля:

— db – раздел с параметрами подключения к базе данных, который используется как первоисточник для выставляемых данных (представленных результатов).

– host – host: порт строка для подключения.

– база данных – строковое значение с именем базы данных.

– collection – строковое значение с названием коллекции.

— config_db – раздел с параметрами подключения к базе данных, которая используется как источник метаданных (например, для получения настроек задачи ML).

– host – host: порт строка для подключения.

– database – строковое значение с именем базы данных.

– collection – строковое значение с названием коллекции.

– config_name – строковое значение имени метаданных (поле config_name из документа с записью метаданных).

— metrics – список метрик.

– name – название метрики;

– precision – необязательный параметр, значение типа int с количеством цифр числа с плавающей запятой после знака «,» отображается в таблице. Значение по умолчанию: 4.

— frontend – раздел с параметрами для отображения полей в GUI (например, оглавление).

– fields – необязательно, список строк с именами полей, которые будут отображаться в виде столбцов в таблице графического интерфейса. По умолчанию все поля имеют числовое форматирование.

– text_fields – необязательно, подмножество полей из полей, которые будут обрабатываться как текстовые поля. Графический интерфейс поддерживает

полнотекстовый поиск в текстовых полях.

– sort_columns – необязательно, по умолчанию список с параметрами сортировки данных в таблице.

• field – строка с полем, ранее описанная в списке полей. • ascending – логическое поле для использования сортировки по возрастанию в таблице по умолчанию.

– searchchable_columns - необязательно, подмножество из text_fields, которое будет использоваться для полнотекстового поиска в графическом интерфейсе.

Запуск сервиса

Чтобы запустить сервис, запустите команду в терминале:

```
./scripts/run_docker_container.sh -p 5001 -t submits_web_app:1.0
↪  -n submits_web_app
```

Параметры контейнера:

- -p – порт для привязки приложения в хост-системе (необязательно);
- -t – тег, который использовался при сборке образа (необязательно);
- -n – имя контейнера (необязательно).

Останов сервиса

Чтобы остановить сервис, используйте интерфейс командной строки докера:

```
docker rm -f {container_id}
```

1.2.4 Metrics Service

Сервис выполняет подсчет значений показателей для сабмитов.

Необходимые условия

Служба работает в контейнерной среде, поэтому базовым условием является настроенный Docker.

Пакеты Docker:

- docker-ce:amd64/bionic 5:18.09.6 или выше
- docker-ce-cli:amd64/bionic 5:18.09.6 или выше

Сборка / установка

Проект содержит скрипт build_docker_image.sh для создания docker-образа службы. Для поддержки адресации каждая сборка получает свой собственный тег сборки CI_TAG.

Шаги сборки / установки:

- создайте новый файл конфигурации, например файл first-metric-service-config.json (см. раздел конфигурации);
- перейдите в корень проекта (он содержит файлы build_docker_image.sh, run_docker_container.sh и т.д.);
- скопируйте файл first-metric-service-config.json в папку config в корне проекта:

```
cp {anywhere-config-was-before}/first-metric-service-config
↪  .json
↪  ./configs/
```

- запустите команду в терминале:

```
./build_docker_image.sh -t $CI_TAG
```

Ваш собственный образ сервиса metrics_service_app создан.

Описание файла конфигурации

Файл конфигурации написан в формате json.

```
{
    "config_name" : "some_task_metrics",
    "config_version" : "0.1",

    "metrics_service_config":
    {
        "port": 27017,
        "host": "10.30.16.181",
        "db": "aero_db_refactored",
        "tasks_to_use_metrics":
        [   "aero__fw_classification_v2",
            "aero__fw_classification_v3"
        ],
        "hashed_fields":[],
        "rebuild_cache":false,
        "update_interval":2,
        "print_progress": false
    },
    "logging":
    {
        "logging_filename": "./logs/metrics_service.log",
        "logging_level": "INFO",
        "copy_logs_to_stdout": false
    }
}
```

Файл конфигурации состоит из нескольких основных полей и двух основных разделов:

metrics_service_config и logging:

– config_name – строковое имя конфигурации сервиса;
– config_version – строковая версия конфигурации сервиса;
– metrics_service_config – раздел с основными настройками сервиса::

- host – строка с хостом службы (например: localhost);
- port – целчисленный тип данных,с сервисным портом (например: 27017);
- db – имя базы данных mongodb с конфигами metrics_service;
- tasks_to_use_metrics – список строк, названия задач, для которых рассчитываются метрики. Задачи необходимо настраивать в db;

- authors_to_check – список строк, белый список авторов, результаты моделирования которых могут быть оценены;
- hashed_fields – список строк, полей, которые будут использоваться при вычислении хеша;
- rebuild_cache – логический флаг для пересчета каждой метрики независимо от значения хэша;
- update_interval – число с плавающей запятой, интервал в секундах для повторения одной итерации основного цикла;
- print_progress – логический флаг для вывода прогресса расчета метрик за один основной цикл в стандартный вывод.

– logging – раздел с настройкой логирования

- logging_filename – строка с путем к файлу журнала;
- logging_level – строка с названием уровня ведения журнала;
- copy_logs_to_stdout – логический флаг для копирования логов сервиса на стандартный вывод.

Файлы конфигурации перемещаются в контейнер на этапе установки, поэтому, чтобы запустить приложение с конкретной конфигурацией, оно должно быть сначала построено с ней внутри.

Запуск сервиса

Чтобы запустить сервиса, требуется ввести команду запуска службы в терминале:

```
./run_docker_container_no_mounts.sh -t $CI_TAG -c
↪  ./configs/first-metric-service-config.json
```

Параметры контейнера:

- -t – тег, который использовался при сборке приложения;
- -c – конфигурация для использования при запуске службы в контейнере;
- -n – суффикс имени контейнера. Общее имя контейнера – metrics_service_$TAG-$NAME.

Суффикс имени необходим для разделения приложений metrics_service между задачами проекта.

Останов сервиса

Чтобы остановить службу, используйте интерфейс командной строки докера:

```
docker rm -f {container_id}
```

1.2.5 Project Wizard

Процесс установки рассмотрен в онлайн-версии документации.

1.2.6 Prediction Builder

Сервис автоматизирует процесс вывода через REST API.

Необходимые условия

Приложение работает в контейнерной среде, поэтому базовым предварительным условием является настроенный https://www.docker.com/докер.

Образ сервиса основан на следующих пакетах докеров:

- openjdk:8 и выше
- docker-ce:amd64/bionic 5:18.09.6 и выше
- docker-ce-cli:amd64/bionic 5:18.09.6 и выше

Сборка / установка

Шаги сборки / установки:

– Этот сервис должен быть построен с использованием https://spring.io/Spring Framework. Он использует автоматическую конфигурацию с использованием файла конфигурации, поэтому сначала вы должны отредактировать файл конфигурации application-compose_api.yml (см. Раздел конфигурации ниже).

– Перейдите в корневой каталог platform-proto-java. Создайте файл .jar с помощью https://gradle.org/gradle:

```
./gradlew :prediction-builder-impl:clean && ./gradlew
↪  :prediction-builder-impl:bootJar
```

– Создайте рабочий каталог, создайте образ докера и запустите контейнер.

```
mkdir -p workdir/logs

cd prediction-builder-impl

docker build -t idcp/prediction-builder-api .

docker run -d \
--name prediction-builder-api \
--mount type=bind,source=${PWD}/../workdir/logs,target=/logs \
-p 8390:8380 \
-e SPRING_PROFILES_ACTIVE='compose_api' \
```

```
-v ${PWD}/src/main/resources/application-compose_api.yml:/applic
 ↪  ation-compose_api.yml
 ↪  \
idcp/prediction-builder-api
```

Также вы можете использовать docker-compose :

```
docker-compose -f docker-compose.<profile>.yml build
 ↪  prediction-builder-api

docker-compose -f docker-compose.<profile>.yml up -d
 ↪  prediction-builder-api
```

– Проверьте статус контейнера:

```
docker ps --filter "name=prediction-builder-api"
```

STATUS Up означает, что контейнер установлен и запущен корректно.

Описание файла конфигурации

Сервис использует файл конфигурации Spring Boot application.yml.

```
    spring:
  profiles:
    active: development
  datasource:
    driver-class-name: org.postgresql.Driver
    url: jdbc:postgresql://localhost:5432/db
    username: <username>
    password: <password>
    hikari:
      minimum-idle: 1
      maximum-pool-size: 4
  jpa:
    hibernate:
      ddl-auto: none
    show-sql: true
    database: postgresql
    database-platform: org.hibernate.dialect.PostgreSQLDialect
    open-in-view: false
    generate-ddl: false
    properties:
      hibernate:
        jdbc:
```

```
          lob:
            non_contextual_creation: true
  jackson:
    serialization:
      write-dates-as-timestamps: true
logging:
  pattern:
    console: "%d %-5level %logger : %msg%n"
  level:
    platform: DEBUG
    org.springframework: INFO
server:
  port: 8380
  sessionTimeout: 30
platform:
  prediction-builder:
    parallelism: 4
  feature-builder:
    endpoint: http://localhost:8280
    export-dir: export
  model-player:
    endpoint: http://localhost:9066
```

Файл конфигурации состоит из следующих строк:

SPRING PROPERTIES

- PROFILES – позволяет определить, какие профили используют эту конфигурацию.

  https://spring.io/Spring Framework – универсальная среда с открытым исходным кодом для платформы Java.

  spring.profiles.active – описывает свойство, определяющее, какие профили активны (строка, например, «разработка»).

- Подключения к текущей базе данных также можно настроить автоматически с помощью пула DATASOURCE

  spring.datasource.driver-class-name – полное имя драйвера JDBC (строка, например, org.postgresql.Driver).

  spring.datasource.url – URL-адрес JDBC базы данных ‘jdbc:postgresql://<hostname>:<port>/<database>‘ (например, ‘jdbc:postgresql://localhost:5432/test-data-api‘).

  spring.datasource.username – логин пользователя базы данных (строка, например, username).

spring.datasource.password – пароль для входа в базу данных (например, password).

spring.datasource.hikari.* – специфические настройки hikari (пул соединений JDBC, готовый к производству с нулевым оверхедом).

- JPA – описывает управление реляционными данными в приложениях с использованием Java.

  spring.jpa.hibernate.ddl-auto – режим DDL. Фактически, это ярлык для свойства «hibernate.hbm2ddl.auto» (например, «none»).

  spring.jpa.show-sql - включить логирование операторов SQL (логический формат данных).

  spring.jpa.database – целевая база данных для работы, по умолчанию определяется автоматически (строка).

  spring.jpa.database-platform – диалект hibernate (инструмент объектно-реляционного сопоставления для Java) (например, «org.hibernate.dialect.»).

  spring.jpa.open-in-view – сеанс гибернации открыт во время обработки HTTP-запроса (логическое значение).

  spring.jpa.generate-ddl – инициализировать схему при запуске (логическое значение).

  spring.jpa.properties.* – дополнительные собственные свойства, устанавливаемые для поставщика JPA.

- JACKSON библиотека для обработки данных JSON на Java

  spring.jackson.serialization.* функции включения / выключения jackson, которые влияют на способ сериализации объектов Java.

LOGGING PROPERTIES

- logging.level.* – сопоставление уровней важности логов. Например, ‘logging.level.org.springframework = INFO‘.

- logging.pattern.console - шаблон аппендера для вывода в консоль. Поддерживается только при настройке входа в систему по умолчанию.

EMBEDDED SERVER CONFIGURATION

- server.port – HTTP-порт сервера (int, например 8390).

- server.sessionTimeout – тайм-аут сеанса в секундах (int, например 30).

PLATFORM PROPERTIES

- platform.prediction-builder.parallelism - количество потоков, работающих в общей неограниченной очереди (целочисленный тип данных, например, 4).
- platform.feature-builder.endpoint – URL-адрес процесса хранилища функций (url, например, http: // localhost: 8590).
- platform.feature-builder.export-dir – каталог экспорта (строка, например, «экспорт»).
- platform.model-player.endpoint – URL-адрес процесса модель-проигрыватель (url, например, http: // localhost: 9066).

Запуск сервиса

Чтобы запустить сервис, выполните команду docker:

```
docker start prediction-builder-api
```

..или используйте docker-compose:

```
docker-compose up -d prediction-builder-api
```

Посла запуска вы можете отправить запрос, например:

```
curl -X POST 'http://localhost:8390/api/prediction_builder/build
↪ ?features_build=1570026953257-3c9422f3-fb82-4c98-a748-2fdf82
↪ f29cde&from=1451606400&to=1451650000&obj=VP-BTP&model=RENAME
↪ D_ENG_2_REV_PRESSURIZED_QAR'
```

Пример ответа сервиса:

```
{
    "predictionsBuilt": 0,
    "predictionsFailed": 0,
    "timeTaken": 0,
    "objects":
        {
            "VP-BTP":
                {
                    "objectId": "VP-BTP",
                    "predictionsBuilt": 0,
                    "predictionsFailed": 0,
```

```
                    "models": {}
                }
        }
}
```

Останов сервиса

Чтобы остановить сервис, используйте интерфейс командной строки докера:

```
docker stop prediction-builder-api
```

..или используйте docker-compose:

```
docker-compose down prediction-builder-api
```

Удаление сервиса

Для удаления сервиса используйте команду docker:

```
docker rm prediction-builder-api
```

..или используйте docker-compose:

```
docker-compose rm prediction-builder-api
```

1.2.7 Feature Store

Приложение предоставляет REST API для работы с функциями.

Необходимые условия

Приложение работает в контейнерной среде, поэтому базовым предварительным условием является настроенный докер.

Образ сервиса основан на следующих пакетах докеров:

- ubuntu:18.04 и выше
- openjdk:11 и выше
- python:3.6 и выше

Сборка / установка

Шаги сборки / установки:

– Этот сервис должен быть построен с использованием https://spring.io/Spring Framework. Он использует автоматическую конфигурацию с использованием файла конфигурации, поэтому сначала вы должны отредактировать файл конфигурации application-compose_api.yml (см. Раздел конфигурации ниже).

– Перейдите в корневой каталог platform-proto-java. Создайте файл .jar с помощью https://gradle.org/gradle:

```
./gradlew :feature-store-impl:clean && ./gradlew
↪  :feature-sore-impl:bootJar
```

– Создайте рабочий каталог, создайте образ докера и запустите контейнер.

```
mkdir -p workdir/logs

cd feature-store-impl

docker build -t idcp/feature-store-api .

docker run -d \
--name feature-store-api \
--mount type=bind,source=${PWD}/../workdir/logs,target=/logs \
-p 8590:8580 \
-e SPRING_PROFILES_ACTIVE='compose_api' \
```

```
-v ${PWD}/src/main/resources/application-compose_api.yml:/applic
 ↪  ation-compose_api.yml
 ↪  \
idcp/feature-store-api
```

Также вы можете использовать docker-compose :

```
docker-compose -f docker-compose.<profile>.yml build
 ↪  feature-store-api

docker-compose -f docker-compose.<profile>.yml up -d
 ↪  feature-store-api
```

– Проверьте статус контейнера:

```
docker ps --filter "name=feature-store-api"
```

STATUS Up означает, что контейнер установлен и запущен корректно.

Описание файла конфигурации

Сервис использует файл конфигурации Spring Boot application.yml.

```
spring:
  profiles:
    active: development
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    database: default
  servlet:
    multipart:
      max-file-size: 2048MB
      max-request-size: 2048MB
logging:
  pattern:
    console: "%d %-5level %logger : %msg%n"
  level:
    platform: DEBUG
    org.springframework: INFO
server:
  port: 8590
  sessionTimeout: 120
platform:
```

```
feature-store:
  type: MergedEntityFeatureStore
  work-dir: "${FEATURE_STORE_PATH:workdir}"
  data-dir: "${FEATURE_STORE_PATH:workdir}/s7/data"
  store-dir: "${FEATURE_STORE_PATH:workdir}/s7/store"
  merge-parallelism: 8
  pool-size: 4
feature-export:
  export-dir: "${FEATURE_STORE_PATH:workdir}/s7/export"
feature-import:
  import-dir: "${FEATURE_STORE_PATH:workdir}/s7/import"
build-info-store:
  type: H2BuildInfoStore
  datasource:
    jdbc-url: jdbc:h2:./${FEATURE_STORE_PATH:workdir}/s7/build
    ↪  _info_db
    username: build_info
    password: build_info
    driver-class-name: org.h2.Driver
    platform: h2
    schema: schema-h2.sql
schedule:
  pool-size: 2
  featureset-cleaner:
    enabled: true
    cron: 0 30 0 * * * # every day in 00:30:00
    featureset-ttl-millis: 2592000000 # one month
  featureset-compressor:
    enabled: true
    cron: 0 0 1 * * * # every day in 01:00:00
    featureset-ttl-millis: 172800000 # two days
```

Файл конфигурации состоит из следующих строк:

SPRING PROPERTIES

- PROFILES – позволяет определить, какие профили используют эту конфигурацию.

  https://spring.io/Spring Framework – универсальная среда с открытым исходным кодом для платформы Java.

  spring.profiles.active – описывает свойство, определяющее, какие профили активны (строка, например, «разработка»).

- JPA – описывает управление реляционными данными в приложениях с использованием Java.

  spring.jpa.hibernate.ddl-auto – режим DDL. Фактически, это ярлык для свойства «hibernate.hbm2ddl.auto» (например, «none»).

  spring.jpa.show-sql - включить логирование операторов SQL (логический формат данных).

  spring.jpa.database – целевая база данных для работы, по умолчанию определяется автоматически (строка).

- SERVLET

  spring.servlet.max-file-size – максимальный размер файла для загрузки (строка, например, 2048 МБ).

  spring.servlet.max-request-size – максимальный размер запроса для загрузки (строка, например, 2048 МБ).

LOGGING PROPERTIES

- logging.level.* – сопоставление уровней важности логов. Например, ‘logging.level.org.springframework = INFO‘.

- logging.pattern.console - шаблон аппендера для вывода в консоль. Поддерживается только при настройке входа в систему по умолчанию.

EMBEDDED SERVER CONFIGURATION

- server.port – HTTP-порт сервера (целочисленное значение, например 8390).

- server.sessionTimeout – тайм-аут сеанса в секундах (целочисленное значение, например 30).

PLATFORM PROPERTIES

- platform.feature-store.type – класс реализации FeatureStore, который используется для хранения функций (строка, «MergedEntityFeatureStore»).

- platform.feature-store.work-dir – путь к рабочему каталогу в проекте (строка, «$ {FEATURE_STORE_PATH: workdir}»).

- platform.feature-store.data-dir - путь к каталогу данных в проекте (строка, -«$ {FEATURE_STORE_PATH: workdir} / s7 / data»).

- platform.feature-store.store-dir – путь к каталогу хранения в проекте (строка, «$ {FEATURE_STORE_PATH: workdir} / s7 / store»).

- platform.feature-store.merge-parallelism – количество потоков, работающих в общей неограниченной очереди (целочисленный, например, 8).
- plaform.feature-store.pool-size - размер основного пула ThreadPoolExecutor (целое число, например, 4). -
- platform.feature-export.export-dir - путь к каталогу экспорта в проекте (строка, «$ {FEATURE_STORE_PATH: workdir} / s7 / export»).
- platform.feature-import.import-dir – путь к каталогу импорта в проекте (строка, «$ {FEATURE_STORE_PATH: workdir} / s7 / import»).

Подключения к текущей базе данных также можно настроить автоматически с помощью пула DATASOURCE.

- platform.build-info-store.type – класс реализации BuildInfoStore, который используется для хранения сборки (строка, «H2BuildInfoStore»)-
- platform.build-info-store.datasource.driver-class-name – полное имя драйвера JDBC (строка, например, «org.h2.Driver»).
- platform.build-info-store.datasource.jdbc-url – URL-адрес JDBC встроенной базы данных (строка, например, ‘jdbc: h2: ./$ {FEATURE_STORE_PATH: workdir} / s7 / build_info_db‘).
- platform.build-info-store.datasource.username – логин пользователя базы данных (строка, например build_info).
- platform.build-info-store.datasource.password – пароль для входа в базу данных (строка, например build_info).
- platform.build-info-store.datasource.platform – платформа для использования в ресурсе схемы (строка, например, h2).
- platform.build-info-store.datasource.schema – ссылка на ресурс сценария схемы (DDL) (схема – $ {platform} .sql, например, «schema-h2.sql»)

- SCHEDULE TASKS

  platform.schedule.pool-size – количество потоков для асинхронного планирования задач (целое число, например 2).

  platform.schedule.feature-build-cleaner.enabled – флаг, разрешающий выполнение чистых сборок (логическое значение).

platform.schedule.feature-build-compress.enabled – флаг для включения выполнения компрессионных сборок (логическое значение).

platform.schedule.feature-build-cleaner.cron – выражение cron для планирования чистых сборок (строка, например, «0 30 0 * * *»).

platform.schedule.feature-build-compress.cron – выражение cron для планирования сборки сжатия (строка, например, «0 0 1 * * *»).

platform.schedule.feature-build-cleaner.build-ttl-millis – время жизни сборок в миллисекундах (тип данных long, например, 2592000000).

platform.schedule.feature-build-compress.build-ttl-millis – время жизни сборок в миллисекундах, а затем их сжатие (тип данных long, например, 2592000000).

Запуск сервиса

Чтобы запустить сервис, выполните команду docker:

```
docker start feature-store-api
```

..или используйте docker-compose:

```
docker-compose up -d feature-store-api
```

После запуска вы можете отправить сервису запрос, например:

```
curl -X POST -F files=@"./feature-store-impl/src/test/resources/
 ↪  test/api/feature/store/test/XX-YYY/features_01.csv"
 ↪  \
'http://localhost:8590/api/feature_builder/import?from=145160640
 ↪  0&to=1454284800'
```

Пример ответа:

```
{"featuresetId":"1571231276940-199d79f4-79bb-4023-bb36-676e51bb6
 ↪  a59","success":true,"imported":1}
```

Останов сервиса

Чтобы остановить приложение, используйте интерфейс командной строки докера:

```
docker stop feature-store-api
```

или используйте docker-compose:

```
docker-compose down feature-store-api
```

Удаление сервиса

Чтобы удалить серсис, используйте интерфейс командной строки докера:

```
docker rm feature-store-api
```

или используйте docker-compose:

```
docker-compose rm feature-store-api
```

1.2.8 Models Player

Сервис автоматизирует жизненный цикл модели машинного обучения.

Необходимые условия

Сервис работает в контейнерной среде, поэтому базовым условием является настроенный Docker.

Сборка / установка

Для создания образа докера используйте скрипт:

```
/scripts/build_docker_image.sh
```

Настройка сервиса

Файл конфигурации написан в формате json. Пример конфигурации по умолчанию представлен ниже:

```
{
    "models_path":"../models",
    "archived_models_path":"../models_archives",
    "start_all_models_on_init":true,
    "shutdown_if_model_start_fail":true,
    "model_start_check_timeout":3,
    "default_models_config": "./configs/config.json"
}
```

Файл конфигурации состоит из следующих строк:

- models_path – путь к каталогу с моделями;
- archived_models_path – путь к каталогу с архивами моделей;
- start_all_models_on_init – флаг логический, если true – все доступные модели будут запущены при запуске службы;
- shutdown_if_model_start_fail – логический флаг, если указано значение true, запускает процессы модели в управляемом режиме – ожидает таймаута после запуска каждого процесса, проверяет код возврата и, при необходимости, удаляет неуспешные процессы и регистрирует события;

- model_start_check_timeout – таймаут для управляемого режима;
- default_models_config – путь по умолчанию к файлу конфигурации модели.

По умолчанию сервис использует конфигурацию из models_player / configs / production.json.

Вместо значений из файла конфигурации вы можете использовать следующие переменные среды. Приоритет значений переменных выше, чем из файла конфигурации.

- HOST
- PORT
- MODELS_PATH
- ARCHIVED_MODELS_PATH
- START_ALL_MODELS_ON_INIT
- SHUTDOWN_IF_MODEL_START_FAIL
- MODEL_START_CHECK_TIMEOUT

Примечание

Если вы запускаете службу в контейнере Docker, хост, порт и все пути являются настройками внутри контейнера. Для установки внешнего порта и хоста используйте флаги команд докеров.

Запуск сервиса

Для старта запустите контейнер Docker:

```
./scripts/run_docker_container.sh
```

Примечание

Текущая версия Models Player использует конфигурацию из config / production.json в корне вашего контекста докера.

Останов сервиса

Чтобы остановить сервис, используйте интерфейс командной строки докера:

```
docker rm -f <container_id>
```

1.2.9 Model Wrapper

Оборачивает модель машинного обучения для совместимости с Models Player.

Установка

Чтобы использовать модель в Models Player, ее нужно «обернуть». Обертка – это представление модели в виде каталога с определенным набором файлов и каталогов внутри:

```
├── model_directory
│ ├── configs
│ │ └── config.json
│ ├── data
│ ├── model_wrapper
│ └── model.py
```

где:

- configs – папка с файлом конфигурации модели;
- data – необязательный каталог, дополнительные данные, необходимые для инициализации модели;
- model_wrapper – папка с содержимым компонента model_wrapper;
- model.py – модуль Python, описывает методы модели init, predict и shutdown.

Примечание

Имя каталога модели используется в Models Player в качестве идентификатора модели (параметр model_id).

Конфигурация модели

Конфигурация обернутой модели записывается в виде json-файла и помещается в директорию configs.

Пример конфигурации модели показан ниже:

```
{
    "model_name": "sum_model",
    "host": "localhost:10005",
```

```
    "model_filename": "sum_model.py",
    "model_class_name": "AllSumModel",
    "message": "Name of formulated problem",
    "model_features": ["feature_1","feature_9","feature_10"],
    "feature_name_aliases":
    {
        "feature_9": "main_feature",
        "feature_10": "feature_3",
    }
    "tags":["is_prod_model"],
    "logging_level": "info"
}
```

Файл конфигурации содержит следующие обязательные поля:

- model_name – название модели, например Ridge, AllSumModel и др .; обратите внимание, что это не идентификатор модели;
- host – адрес (хост и порт) для сервера gRPC с упакованной моделью;
- model_filename – имя файла Python, который содержит класс с методами инициализации, прогнозирования и завершения модели;
- model_class_name – имя класса Python, который используется для описания модели в модуле <model_filename>;
- message – название текущей сформулированной проблемы;
- model_features – список функций, используемых моделью;
- feature_name_aliases – словарь с псевдонимами функций в формате {true_name: alias}; после применения псевдонимов старые имена теряются;
- tag - ключевые слова для выбора способа работы с моделью, например, тег is_prod_model означает, что модель должна использоваться в производстве;
- logging_level – уровень ведения журнала, может быть info", "warn", "error" и т. д.

Перечисленные поля обязательны, но вы всегда можете добавить любые дополнения в этот файл конфигурации.

Формат класса модели

Модуль, описывающий работу модели машинного обучения.

Модуль должен соответствовать следующим правилам:

- он должен иметь класс с именем, описанным в конфигурационном файле модели (поле model_class_name field);
- класс должен быть наследником ModelAbstractWrapper;
- класс должен иметь методы model_init, предсказать и выключить с инструкциями по инициализации, делать прогнозы и останавливать модель соответственно.

Пример класса модели (составить сумму входных значений и вернуть ее в качестве прогноза):

```
import numpy as np
import sys, os

sys.path.append(os.path.dirname(os.path.realpath(__file__)))
from model_wrapper.src.model_abstract_wrapper import
 ↪  ModelAbstractWrapper

class AllSumModel(ModelAbstractWrapper):
    def model_init(self, model_parameters=dict()):
        pass

    def predict(self, X):
        return np.sum(X, axis=1)

    def shutdown(self):
        pass
```

Запуск и остановка службы

Запуск и остановка моделей осуществляется через служебный интерфейс Models Player (см. документ ”Описание интерфейсов”).

1.2.10 Feature Builder

Сервис автоматизирует процесс сборки функций через REST API.

Необходимые условия

Приложение работает в контейнерной среде, поэтому базовым предварительным условием является настроенный https://www.docker.com/докер. Образ сервиса основан на следующих пакетах докеров:

- ubuntu:16.04 или выше
- openjdk:8 или выше
- python:3.6 или выше

Сборка / установка

Шаги сборки / установки:

– Этот сервис должен быть построен с использованием Spring Framework. Он использует автоматическую конфигурацию с использованием файла конфигурации, поэтому сначала вы должны отредактировать файл конфигурации application-compose_api.yml (см. Раздел конфигурации ниже).

– Перейдите в корневой каталог platform-proto-java. Создайте файл .jar с помощью https://gradle.org/gradle:

```
./gradlew :feature-builder-impl:clean && ./gradlew
↪  :feature-builder-impl:bootJar
```

– Создайте рабочий каталог, создайте образ докера и запустите контейнер.

```
mkdir -p workdir/logs

mkdir -p workdir/aero_s/extractors

mkdir -p workdir/aero_s/dump

cd feature-builder-impl

docker build -t idcp/feature-builder-api .

docker run -d \
--name feature-builder-api \
```

```
--mount type=bind,source=${PWD}/../workdir/logs,target=/logs \
-p 8290:8280 \
-e SPRING_PROFILES_ACTIVE='compose_api' \
-v ${PWD}/src/main/resources/application-compose_api.yml:/applic
 ↪  ation-compose_api.yml
 ↪  \
idcp/feature-builder-api
```

…или можно использовать docker-compose :

```
docker-compose -f docker-compose.<profile>.yml build
 ↪  feature-builder-api

docker-compose -f docker-compose.<profile>.yml up -d
 ↪  feature-builder-api
```

– Проверьте статус контейнера:

```
docker ps --filter "name=feature-builder-api"
```

STATUS Up означает, что контейнер установлен и запущен корректно.

Описание файла конфигурации

Сервис использует файл конфигурации Spring Boot application.yml.

```
spring:
  profiles:
    active: development
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
    database: default
logging:
  pattern:
    console: "%d %-5level %logger : %msg%n"
  level:
    platform: DEBUG
    org.springframework: INFO
server:
  port: 8280
  sessionTimeout: 120
```

```
platform:
  data-client:
    type: DataClientAdapter
    endpoint: http://localhost:8170
    namespace: <namespace>
  feature-extractor:
    type: FeatureExtractorCmd
  feature-builder:
    pipeline:
      loader: ExtractorLoaderImpl
      starter: ExtractorStarterImpl
      filter: ExtractorResultFilterImpl
    extractors-dir:
     ↪  "${FEATURE_STORE_PATH:workdir}/aero_s/extractors"
    data-dir: "${FEATURE_STORE_PATH:workdir}/aero_s/data"
    built-dir: "${FEATURE_STORE_PATH:workdir}/aero_s/built"
    parallelism: 4
    limit: 2
    pool-size: 4
  meta-store:
    type: H2MetaStore
    datasource:
      jdbc-url: jdbc:h2:./${FEATURE_STORE_PATH:workdir}/aero_s/m⌋
       ↪  eta_store_db
      username: meta_store
      password: meta_store
      driver-class-name: org.h2.Driver
      platform: h2
      schema: schema-h2.sql
  feature-store:
    endpoint: http://localhost:8590
```

Файл конфигурации состоит из следующих строк:

- **SPRING PROPERTIES**

  PROFILES - позволяет определить, какие профили используют эту конфигурацию. https://spring.io/Spring Framework – универсальная среда с открытым исходным кодом для платформы Java.

  spring.profiles.active – описывает свойство, указывающее, какие профили активны (строка, например, ”development”).

  JPA – описывает управление реляционными данными в приложениях с использованием Java.

spring.jpa.hibernate.ddl-auto – режим DDL. Фактически это ярлык для свойства «hibernate.hbm2ddl.auto» (строка, например, none).

spring.jpa.show-sql - включить логирование операторов SQL (логическое значение).

spring.jpa.database - целевая база данных для работы, автоматически определяется по умолчанию (строка).

- **LOGGING PROPERTIES**

logging.level.* – сопоставление уровней важности логов. Например, ‘logging.level.org.springframework = INFO‘.

logging.pattern.console – шаблон аппендера для вывода в консоль. Поддерживается только при настройке входа в систему по умолчанию.

- **EMBEDDED SERVER CONFIGURATION**

server.port – HTTP-порт сервера (целочисленное значение, например, 8290).

server.sessionTimeout – тайм-аут сеанса в секундах (целочисленное значение, например 30).

- **PLATFORM PROPERTIES**

platform.data-client.type – класс реализации DataClient, который используется для предоставления доступа к службам данных (строка, «DataClientAdapter»).

platform.data-client.endpoint – URL-адрес процесса службы данных (URL, например, http: // localhost: 8170).

platform.data-client.namespace – имя базы данных (строка, например, «aerophm»).

platform.feature-extractor.type – класс реализации FeatureExtractor, который используется для извлечения функций (строка, «FeatureExtractorCmd»).

platform.feature-builder.pipeline. * – классы реализации конвейера построителя, которые определяют этапы построения функций.

platform.feature-builder.extractors-dir – путь к каталогу экстракторов в проекте (строка, «$ {FEATURE_STORE_PATH: workdir} / aero_s / extractors»).

platform.feature-builder.data-dir – путь к каталогу данных в проекте (строка, «$ {FEATURE_STORE_PATH: workdir} / aero_s / data»).

platform.feature-builder.built-dir – путь к встроенному каталогу в проекте (строка, «$ {FEATURE_STORE_PATH: workdir} / aero_s / built»).

platform.feature-builder.parallelism – количество потоков, работающих в общей неограниченной очереди (целое значение, например, 4).

platform.feature-builder.limit – ограничение количества операций (целое значение, рекомендуется 2).

plaform.feature-builder.pool-size – размер основного пула (целое число, например, 4).

platform.feature-store.endpoint – URL-адрес процесса хранилища функций (url, например, http: // localhost: 8590).

Подключения к производственной базе данных также можно настроить автоматически с помощью пула DATASOURCE.

platform.meta-store.datasource.driver-class-name – полное имя драйвера JDBC (строка, например, «org.h2.Driver»).

platform.meta-store.datasource.jdbc-url – URL-адрес JDBC встроенной базы данных (строка, например, ‘jdbc: h2: ./$ {FEATURE_STORE_PATH: workdir} / aero_s / meta_store_db‘).

platform.meta-store.datasource.username – логин пользователя базы данных (строка, например, meta_store).

platform.meta-store.datasource.password – пароль для входа в базу данных (строка, например, meta_store).

platform.meta-store.datasource.platform – платформа для использования в ресурсе схемы (строка, например, h2).

platform.meta-store.datasource.schema – ссылка на ресурс сценария схемы (DDL) (схема - $ {platform} .sql, например, «schema-h2.sql»).

Запуск сервиса

Чтобы запустить службу, выполните команду docker:

```
docker start feature-builder-api
```

..или используйте docker-compose:

```
docker-compose up -d feature-builder-api
```

После запуска вы можете отправить запрос в сервис, например:

```
curl -X POST -F files=@"./feature-builder-impl/src/test/resource⌋
 ↪  s/test/api/feature/store/test/XX-YYY/features_01.csv"
 ↪  \
'http://localhost:8290/api/feature_builder/import?from=145160640⌋
 ↪  0&to=1454284800'
```

Пример ответа:

```
{"buildId":"1571231276940-199d79f4-79bb-4023-bb36-676e51bb6a59",
↪  "success":true,"imported":1}
```

Останов сервиса

Чтобы остановить сервис, используйте интерфейс командной строки докера:

```
docker stop feature-builder-api
```

..или используйте docker-compose:

```
docker-compose down feature-builder-api
```

Удаление сервиса

Чтобы удалить службу, используйте команду docker:

```
docker rm feature-builder-api
```

..или используйте docker-compose:

```
docker-compose rm feature-builder-api
```

1.2.11 Data Service

Сервис предоставляет REST API для запросов к хранилищу данных.

Необходимые условия

Приложение работает в контейнерной среде, поэтому базовым предварительным условием является настроенный https://www.docker.com/докер.

Образ службы основан на следующих пакетах докеров:

- ubuntu:16.04 или выше
- openjdk:8 или выше
- python:3.6 или выше

Сборка / установка

Шаги сборки / установки:

– Этот сервис должен быть построен с использованием https://spring.io/Spring Framework. Он использует автоматическую конфигурацию с использованием файла конфигурации, поэтому сначала вы должны отредактировать файл конфигурации application-compose_api.yml (см. раздел конфигурации ниже).

– Перейдите в корневой каталог platform-proto-java. Создайте файл .jar с помощью https://gradle.org/gradle:

```
./gradlew :data-service-impl:clean && ./gradlew
↪  :data-service-impl:bootJar
```

– Создайте каталог для журналов (в этом примере workdir / logs), создайте образ докера и запустите контейнер.

```
mkdir -p workdir/logs

cd data-service-impl

docker build -t idcp/data-service-api .

docker run -d \
--name data-service-api \
--mount type=bind,source=${PWD}/../workdir/logs,target=/logs \
-p 8170:8170 \
```

```
-e SPRING_PROFILES_ACTIVE='compose_api' \
-v ${PWD}/application-compose_api.yml:/application-compose_api.y
 ↪  ml
 ↪  \
idcp/data-service-api
```

…или используйте docker-compose :

```
docker-compose -f docker-compose.<profile>.yml build
 ↪  data-service-api

docker-compose -f docker-compose.<profile>.yml up -d
 ↪  data-service-api
```

– Проверьте статус контейнера:

```
docker ps --filter "name=data-service-api"
```

STATUS Up означает, что контейнер успешно установлен и запущен.

Описание файла конфигурации

Сервис использует файл конфигурации Spring Boot application.yml.

```
spring:
  profiles:
    active: development
  datasource:
    driver-class-name: org.postgresql.Driver
    url:
     ↪  jdbc:postgresql://localhost:5432/${dataservice.namespace}
    username: <username>
    password: <password>
    hikari:
      minimum-idle: 1
      maximum-pool-size: 4
dataservice:
  data-catalog: DataCatalogPSQLImpl
  data-query: DataQueryImpl
  namespace: <namespace>
  schema: public
  proxy: psql
server:
```

```
  port : 8170
logging:
  level:
    platform: DEBUG
    org.springframework: INFO
    org.hibernate: ERROR
```

Файл конфигурации состоит из следующих строк:

- **SPRING PROPERTIES**

  PROFILES – позволяет определить, какие профили используют эту конфигурацию.

  spring.profiles.active – описывает свойство, определяющее, какие профили активны (строка, например, ”development”).

  Подключения к текущей базе данных также можно настроить автоматически с помощью пула DATASOURCE.

  spring.datasource.driver-class-name – полное имя драйвера JDBC (строка, например, “org.postgresql.Driver”).

  spring.datasource.url – URL-адрес JDBC базы данных ‘jdbc:postgresql://<hostname>:<port>/$dataservice.namespace‘ (URL, например, ‘jdbc:postgresql://localhost:5432/$dataservice.namespace‘).

  spring.datasource.username – логин пользователя базы данных (строка, например, “username”).

  spring.datasource.password – пароль для входа в базу данных (например, “password”).

  spring.datasource.hikari.* - специальные настройки для hikari.

- **DATASERVICE CONFIGURATION**

  dataservice.data-catalog -DataCatalogPSQLImpl – имя компонента сервиса каталога данных.

  dataservice.data-query -DataQueryImpl – имя компонента сервиса запроса данных.

  dataservice.namespace – имя базы данных (строка, например, “test-data-api”).

  dataservice.schema – имя схемы (строка, например, “public”).

  dataservice.proxy – имя прокси (строка, например, “psql”).

- **EMBEDDED SERVER CONFIGURATION**

  server.port – HTTP-порт сервера (целочисленное значение, например, 8190).

- **LOGGING PROPERTIES**

  logging.level.* – сопоставление уровней важности логов. Например, ‘logging.level.org.springframework=INFO‘.

Запуск сервиса

Для запуска сервиса, выполните команду docker:

```
docker start data-service-api
```

..или используйте docker-compose:

```
docker-compose up -d data-service-api
```

После запуска вы можете отправить запрос сервису, например:

```
curl 'http://127.0.0.1:8170/datasets'
```

Пример ответа:

```
[
    {"namespace":"aerophm","name":"aids_reports_01","versions":[
     ↪  "1.0"]},
    {"namespace":"aerophm","name":"aids_reports_02","versions":[
     ↪  "1.0"]},
    {"namespace":"aerophm","name":"aircrafts","versions":["1.0"]}
]
```

Останов сервиса

Чтобы остановить службу, используйте интерфейс командной строки докера:

```
docker stop data-service-api
```

..или используйте docker-compose:

```
docker-compose down data-service-api
```

Удаление сервиса

Чтобы удалить сервис, используйте команду docker

```
docker rm data-service-api
```

..или используйте docker-compose:

```
docker-compose rm data-service-api
```

1.2.12 Subject Domain Cache Service

Сервис предоставляет возможность кэшировать различные пары ключ-значение в базе данных Redis с помощью REST API.

Необходимые условия

Приложение работает в контейнерной среде, поэтому базовым предварительным условием является настроенный https://www.docker.com/докер.

Сборка / установка

Проект содержит скрипт scripts/deploy_subject_domain_cache.sh для построения docker-образа службы. Запустите его для развертывания службы:

```
./scripts/deploy_subject_domain_cache.sh
```

Сервис развернут.