Описание

Модуль работы с СУБД

Общие настройки

Подключение к БД через настройки application.yml

```
jprime:
  storage:
    <уникальный код хранилища>:
      title: <описание хранилища>
      type: db
      driver: <jdbc драйвер>
      url: <урл подключения>
      username: <логин>
      password: <пароль>
      transactionIsolation: <уровень изоляции>
      aliases:
        <синоним 1>: <описание хранилища>
        <синоним 2>: <описание хранилища>
        <синоним 3>: <описание хранилища>
```

Основной принцип

Для построения взаимодействия с СУБД через JPrime API и метаописание используется `querydsl-sql`

Хранилища

Пул соединения

На каждое хранилище создается отдельный пул соединений.

При использовании алиасов можно описать дополнительные хранилища с теми же настройками, что и основное. Пул соединений в это случае будет общий

```
jprime:
  storage:
    test1:
      title: Хранилище 1
      type: db
      driver: <jdbc драйвер>
      url: <урл подключения>
      username: <логин>
      password: <пароль>
      aliases:
        test2: Хранилище 2
```

При необходимости разделить пулы соединений рекомендуется описать два разных хранилища, даже, если они используют одну и ту же БД

```
jprime:
  storage:
    test1:
      title: Хранилище 1
      type: db
      driver: <jdbc драйвер>
      url: <урл подключения>
      username: <логин>
      password: <пароль>

    test2:
      title: Хранилище 2
      type: db
      driver: <jdbc драйвер>
      url: <урл подключения>
      username: <логин>
      password: <пароль>
```

Получение хранилища по коду

```
   @Autowired
  private DbRepositoryStorage dbStorage;
  ....
  DbStorage dbStorage = dbStorage.getStorage(<уникальный код хранилища>);
```

Проверка наличия таблицы в хранилище

```
   dbStorage.isTableExists(<имя таблицы>);
```

Проверка наличия колонки в таблице

```
   dbStorage.isColumnExists(<имя таблицы>, <имя колонки>);
```

Получение JdbcTemplate для прямого построения запроса

```
   JdbcTemplate jdbcTemplate = dbStorage.getJdbcTemplate();


  NamedParameterJdbcTemplate namedParameterJdbcTemplate = dbStorage.getNamedJdbcTemplate();
```

Построение сложных запросов

Для построения сложных запросов (с join и пр.) используется API QueryDsl

Пример:

```$java rep.query(MainStorage.CODE, (qSupport, template) -> { QJPClass jpClass1 = qSupport.from(<код класса 1>); QJPClass jpClass2 = qSupport.from(<код класса 2>);
```

```
   ComparablePath jpAttr_2_1 = jpClass1.get(<код атрибута 2 класса 1>);
  ComparablePath jpAttr_3_1 = jpClass1.get(<код атрибута 3 класса 1>);
  ComparablePath jpAttr_1_2 = jpClass2.get(<код атрибута 1 класса 2>);
  ComparablePath jpAttr_2_2 = jpClass2.get(<код атрибута 2 класса 2>);

  SQLQuery<Tuple> query = template.getSQLQuery().select(jpAttr_1_2, jpAttr_3_1)
      .from(jpClass1)
      .innerJoin(jpClass2).on(jpAttr_2_1.eq(jpAttr_1_2))
      .where(new BooleanBuilder()
          .and(jpAttr_1_2.eq(parser.parseTo(Long.class, id.getId())))
          .and(jpAttr_2_2.eq(0))
      );
  Tuple t = query.fetchFirst();
  Object jpAttr_1_1_Val = t.get(jpAttr_1_1);
}
```

);

```
### Массовая вставка/обновление объектов

Batch обработка большого количества объектов возможна при использовании возможностей `SQLInsertClause`
или `SQLUpdateClause`

Пример:

```$java
repo.query(<код хранилища>, (qSupport, template) -> {
    QJPClass metaClass = qSupport.from(<код класса>);
    SQLInsertClause insertTemplate = template.getSQLInsertClause(metaClass);
    for (JPObject o : objs) {
      insertTemplate.set(<атрибут 1>, <значение>);
      insertTemplate.set(<атрибут 2>, <значение>);
      ....
      insertTemplate.addBatch();
    }
    insertTemplate.execute();
  }
}
```

Также возможна вставка через `JPBatchCreate`

**Важно! В каждом батче должны быть одинаковые атрибуты (не путать со значением).**

Пример:

```$java JPBatchCreate.Builder builder = JPBatchCreate.create(<код класса>) .source() .auth();

collection.forEach(object -> builder .set(<атрибут 1>, <значение>) .set(<атрибут 2>, <значение>) ....
addBatch());

repo.batch(builder.build());
```
```

```
## Мониторинг времени выполнения долгих запросов

Включается опцией `jprime.dataaccess.jdbc.query.executeTimeout.alarm.enabled=true`

Допустимый период выполнения запроса `jprime.dataaccess.jdbc.query.executeTimeout.alarm.limitSeconds`,
по умолчанию 30 секунд,

при превышении которого в логе будет указана ошибка с кодом `jdbc.query.queryTimeoutExceeded`
```

jdbc.query.queryTimeoutExceeded - Execution time is 00:12:03. SQL: select count(*) from pers pers where to_tsquery('russian',?) @@ pers.name_tsv

```
## Динамические свойства объекта

Метакласс `jpFeature`

### Хранилище

Имеет код `jpFeatures` и настраивается отдельно

### Применение

Используется при ограничениях запросах вида
```

Filter.feature(<код>).check(<дата>)

```
при выполнении встраивается в тело запроса в виде
```

AND

```
при этом в SQL запроса используются следующие константы:

* OBJECTID - поле идентификатор мета класса
* CHECKDAY - значение дня, на которую идет проверка (для типа свойства = 1)
* CHECKFROMDAY - значение дня, с которого идет проверка (для типа свойства = 2)
* CHECKTODAY - значение дня, по которое идет проверка (для типа свойства = 2)
* AUTH_USERID - Идентификатор пользователя
* AUTH_ORGID - Организация пользователя

Поля:
```

- Код
- Мета код класса
- Название
- Тип
- Описание
- SQL для получения свойства
- Дата создания
- Дата изменения

```
Пример:

| Свойство     | Значение
| ------------ | ------------------
| Код | pers_date_n_eq
| Мета код класса | pers
| Название | назначенные на прием
| Тип | 1
| Описание | находим граждан (pers), у которых дата следующей явки (pers.date_n) = дате фильтрации;
| SQL | {OBJECTID} IN (select t.reg_num from pers t where t.date_n::date = {CHECKDAY}::date)

## SQL запрос

Метакласс `jpQuery`

## Хранилище

Имеет код `jpQuery` и настраивается отдельно

### Применение

Используется в произвольных частях системы для выполнения предсохраненных запросов, полученных по коду
```

@Autowired private JPQueryService jpQueryService; .... String sql = jpQueryService.getSQL(<уникальный код запроса>); ```