Базовый модуль JPrime

Содержит глобальные константы и API для межсервисного взаимодействия

Системные события

Все события в системе являются асинхронными.

Событие является наследником класса `mp.jprime.events.systemevents.JPSystemEvent` и может быть инициировано в любом сервисе системы

События инициируются и обрабатываются программным кодом и используются для передачи информации между сервисами

Инициализация события

```
 private SystemEventPublisher eventPublisher;

@Autowired(required = false)
private void setSystemEventPublisher(SystemEventPublisher eventPublisher) {
  this.eventPublisher = eventPublisher;
}

...

private void publishEvent(String code) {
  if (eventPublisher == null) {
    return;
  }
  eventPublisher.publishEvent(new <Event extends JPSystemEvent>());
}
```

Подписывание на событие

Класс-обработчик должен содержать метод с логикой обработки события

```
 @EventListener
public void handleApplicationEvent(JPSystemApplicationEvent event) {
  JPSystemEvent jpSystemEvent = event.getEvent();
  ....
}
```

Реализация SystemEventPublisher

Существует две реализации передачи системных событий

- `KafkaSystemEventService` . Поддержка обмена событиями через Kafka.

Включается опцией `jprime.events.systemevents.kafka.enabled=true` и рекомендуется для микросервисных сборок

- `ApplicationSystemEventService` . Поддержка обмена событиями внутри приложения

Включается опцией `jprime.events.systemevents.app.enabled=true` и рекомендуется для монолитных сборок

Глобальные события

Глобальные события — это системные события для общения между разными подсистемами.

Событие является наследником класса `mp.jprime.events.systemevents.JPSystemEvent` и может быть инициировано в любом сервисе любой подсистемы.

> ВАЖНО! В подсистеме за чтение топика глобальных событий должен отвечать только один докер! Чтобы гарантировать доставку, слушатель глобальных событий настроен на чтение `eariest`, поэтому у него должен быть статичный неслучайный `group_id`. Поэтому во избежание коллизий включать слушателя глобальных событий необходимо только в одном сервисе внутри контура.

Адресация событий

Атрибуты `producer` и `consumer` опциональны. При необходимости можно адресовать событие в определенную подсистему, передав её код в `consumer`. Если переданный `consumer` пустой, то событие обрабатывается всеми подсистемами, в которых подключен слушатель глобальных событий.

Код `consumer` из события сравнивается с кодом текущей системы, который получается из `JPAppProperty`. По умолчанию он совпадает с кодом текущего jprime сервиса (подходит для монолитной архитектуры).

В случае микросервисной архитектуры рекомендуется создать проектный наследник `AppProperty`, возвращающий код подсистемы, и пометить его аннотацией `@Primary`.

```
@Service
@Primary
@Lazy(value = false)
public class EguintegAppProperty extends AppProperty {

  private static final String SYSTEM_CODE = "my-system-code";

  @Autowired
  public EguintegAppProperty(Environment environment) {
    super(environment);
  }

  /**
   * Код подсистемы
   *
   * @return Код подсистемы
   */
  @Override
  public String systemCode() {
    return SYSTEM_CODE;
  }
}
```

Инициализация события

```
 private GlobalEventPublisher eventPublisher;

@Autowired(required = false)
private void setSystemEventPublisher(GlobalEventPublisher eventPublisher) {
  this.eventPublisher = eventPublisher;
}

...

private void publishEvent(String code) {
  if (eventPublisher == null) {
    return;
  }
  eventPublisher.publishEvent(new <Event extends JPSystemEvent>());
}
```

Подписывание на событие

Класс-обработчик должен содержать метод с логикой обработки события

```
 @EventListener
public void handleApplicationEvent(JPSystemApplicationEvent event) {
  JPSystemEvent jpSystemEvent = event.getEvent();
  ....
}
```

Динамические слушатели Кафки

Этот механизм позволяет динамически создавать, масштабировать, запускать, останавливать слушателей. В том числе создавать каскады слушателей, задав основной топик и настройки каскада (см. реализацию Dead Letter Queue).

Динамические слушатели представлены интерфейсом `JPKafkaDynamicConsumer` и его базовой реализацией `JPKafkaDynamicBaseConsumer<K, V>`. Базовая логика работы с ними описана в абстрактном классе `JPKafkaDynamicConsumerBaseService<K, V>`. Для реализации предлагается создать наследника этого класса или другого более специфичного наследника базового класса.

Настройки слушателей

Создав наследника `JPKafkaDynamicConsumerBaseService<K, V>`, можно задать настройки путём переопределения соответствующих методов. Некоторые из них являются абстрактными и обязательны для реализации, другие же имеют реализацию по умолчанию.

Метод	По умолчанию	Описание
getAutoOffsetReset()	`earliest`	Задает настройку, как слушателю следует читать топик, если его оффсет неизвестен. Возможны варианты `earliest`, `latest`
getMaxPollRecords()	1	Настройка количества событий, читаемых из топика за раз

Метод	По умолчанию	Описание
getMaxPollInterval()	300_000	Максимальная задержка между вызовами poll() слушателем (мс)
getGroupId()	-	Идентификатор слушателя
getBootstrapAddress()	-	Адрес кафки
getKeyDeserializer()	-	Конвертер ключа события
getValueDeserializer()	-	Конвертер значения события
getConsumerConcurrency()	1	Количество параллельных слушателей топика

Dead Letter Consumers

Один из вариантов использования динамических слушателей - это реализация паттерна `Dead Letter Queue`. `JPKafkaDeadLetterConsumerService<K, V>` решает проблему необходимости динамического создания `1..N` слушателей согласно настройкам.

В настройках задается только адрес Кафки, основной топик и настройки частоты обработки событий из дублирующих топиков. При этом сколько настроек повторных попыток будет передано, столько будет создано слушателей в каскаде (и соответственно будет создано `N-1` топиков восстановления).

При исчерпании попыток обработки события (согласно настройкам) оно перекладывается в следующий топик из каскада. Если текущий слушатель последний, то событие перекладывается в конец очереди текущего (последнего) топика. При необходимости можно переопределить метод `protected ConsumerRecordRecoverer getRecoverer(KafkaOperations<K, V> template, String recoveryTopic)`, например, чтобы дифференцировать поведение в зависимости от вида ошибки.

Количество топиков в каскаде и соответствующие интервалы опроса каждого из топиков каскада задаются имплементацией метода `protected abstract Collection<Long> getPollIntervals()`. Если метод вернет пустую коллекцию, то будет создан только один слушатель основного топика с `pollInterval = 0`.

- Пример:

```
mp:
  jprime:
    application-update:
      kafkaServers: 172.16.1.171:9092
      kafkaTopic: egu-application-update
      dead-letter-queue-settings: 0,30000,60000  # 3 слушателя в каскаде: основной + 2 дополнительных
```

```
class MyClass extends JPKafkaDeadLetterConsumerService<String, String> {

  @Value("#{'${mp.jprime.application-update.dead-letter-queue-settings:5}'.split(',')}")
  private List<Long> pollIntervals;

  /**
   * Интервалы опроса топиков каскада (мс)
   */
  @Override
  protected Collection<Long> getPollIntervals() {
    return pollIntervals;
  }
}
```

> ВАЖНО! Если уменьшать количество слушателей в каскаде (было 5 - перезапустили с настройками для 3), возникает риск потери событий, которые уже были положены в топики, для которых согласно новым настройкам уже нет слушателей. Эта проблема решена путем создания слушателей по умолчанию для тех топиков, для которых выполняются условия: * топик подходит по паттерну `<основной топик каскада>-<N>` * для топика ещё нет слушателя в каскаде
>
> Для этих слушателей существуют настройки на уровне `jprime`. При настройке `infinite = true` события, при обработке которых получена ошибка, перекладываются в конец текущей очереди. Если `infinite = false`, то они перекладываются в основной топик, чтобы постепенно перевести все события в актуальный каскад топиков.

Настройка	Значение по умолчанию	Описание
jprime.kafka.dead-letter-consumers.poll-interval	60000	Интервал в миллисекундах между повторными попытками обработки в миллисекундах
jprime.kafka.dead-letter-consumers.infinite	false	Если задано `true`, то при получении ошибки событие перекладывается в конец той же очереди, иначе перекладывается в основной топик, чтобы постепенно перевести все события в актуальный каскад топиков

Формат вывода ошибок при http-запросах

- при 4** или 5** статусах в теле ответа дополнительно содержится детализация ошибок в виде массива:

```
{
  "status": 400,
  "error": "Bad Request",
  ...
  "details": [
    {
      "code": "chairsNeat.some_number.error1",
      "message": "Укажите вещественное число менее 100"
    },
    {
      "code": "chairsNeat.some_number.error2",
      "message": "Ошибка для количества"
    }
  ]
}
```

- дополнительно при 409 ошибке может содержаться логин и описание пользователя, изменившего ресурс

```
{
    ...,
    "details": [
        {
            "code": "jpReportDBRepository.update.versionNotFound",
            "message": "Версия объекта с кодом \"Sosiska\" не соответствует актуальной",
            "updatedUserDescription": "Ульянов Сергей",
            "updatedUserLogin": "sulyanov"
        }
    ]
}
```

- если конкретизировать ошибку не представляется возможным, то детализация выглядит следующим образом

```
{
    ...,
    "details": [
        {
            "code": "server.error",
            "message": null
        }
    ]
}
```