

**САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ
ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО**

Дисциплина: Бэк-энд разработка

Отчет

Лабораторная работа №6

Выполнил:

Фирсов Илья

К3441

Проверил:

Добряков Д. И.

Санкт-Петербург

2026 г.

Задача

По выбранному варианту необходимо будет реализовать межсервисное взаимодействие посредством очередей сообщений.

Ход работы

1. Анализ бизнес-процессов

- Определение двух режимов работы системы
- Выделение точек асинхронного взаимодействия
- Проектирование потоков сообщений

2. Проектирование MQ архитектуры

- Выбор типа exchange (topic, direct, headers)
- Определение routing keys
- Проектирование durable queues

3. Реализация publisher-subscriber паттерна

- Publisher компоненты
- Consumer компоненты
- Обработка сообщений и acknowledgments

4. Отказоустойчивость и масштабируемость

- Dead letter queues
- Consumer pools
- Load balancing

Описание хода работы

1. Бизнес-контекст и два режима работы

Проект CoolThing реализует автоматизацию маркетинга в Telegram с двумя основными режимами работы:

Scripts Mode (Готовые сценарии)

Администратор создает вариативные текстовые сценарии через веб-интерфейс, которые затем исполняются ботами в целевых чатах.

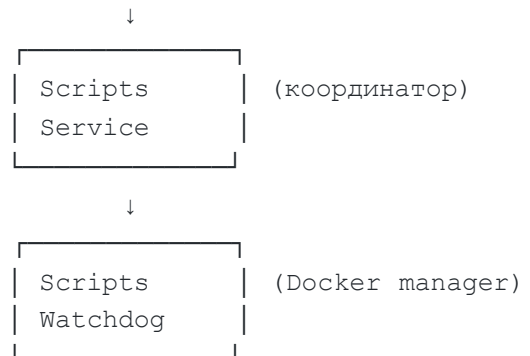
Parser Mode (Активный поиск)

Боты-парсеры мониторят множество чатов на ключевые слова, при обнаружении целевой аудитории менеджеры предлагают услуги через LLM в личных сообщениях.

2. Scripts Mode - Publisher-Subscriber паттерн

Архитектура взаимодействия

Admin Service → [NewActiveScript] → Scripts Exchange → Multiple Consumers



Publisher: Admin Service

```
# admin/infrastructure/mq/__init__.py
class RabbitMQEventsRepository(EventsRepositoryInterface):
    async def publish(self, event: BaseEvent):
        exchange = await channel.declare_exchange('scripts_exchange', ExchangeType.TOP)
        routing_key = f"admin.script_created"
        await exchange.publish(
            aio_pika.Message(body=json.dumps(event, cls=EnhancedJSONEncoder).encode())
            routing_key=routing_key
        )
```

Consumers: Scripts и Scripts Watchdog

```
# scripts/infrastructure/mq/consumer.py & scripts_watchdog/infrastructure/mq/consumer.
class RabbitListener:
    async def _consume(self):
        exchange = await channel.declare_exchange('scripts_exchange', ExchangeType.TOP)
        queue = await channel.declare_queue('scripts_queue')
        await queue.bind(exchange, routing_key='admin.script_created')

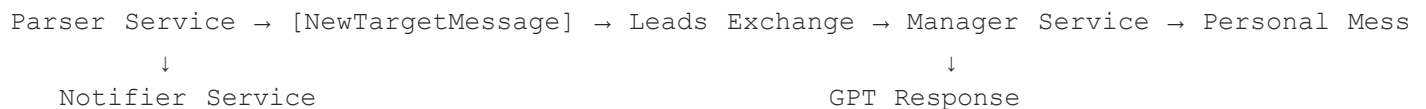
    async with queue.iterator() as queue_iter:
        async for message in queue_iter:
            await self.callback(message)
            await message.ack()
```

Поток выполнения скрипта

1. Admin публикует NewActiveScript в exchange
2. Scripts service получает сообщение, подготавливает данные для выполнения
3. Scripts watchdog получает сообщение, создает Docker контейнер scripts_worker
4. Scripts worker выполняет сценарий в Telegram
5. Scripts watcher отслеживает статус через API

3. Parser Mode - Event-Driven паттерн

Архитектура взаимодействия



Publisher: Parser Service

```
# parser/infrastructure/rabbitmq/event.py
class RabbitMQEventRepository(EventRepository):
    async def publish_target_message(self, message: NewTargetMessage):
        # Публикация в leads exchange с routing key по приоритету
        await self._publish_to_exchange(
            exchange_name='leads_exchange',
            routing_key=f'lead.{message.campaign_id}',
            message=message
        )
```

Consumer: Manager Service

```
# manager/infrastructure/rabbit.py
class RabbitListener:
    async def start_consuming(self):
        exchange = await channel.declare_exchange('leads_exchange', ExchangeType.TOPIC)
        queue = await channel.declare_queue('', exclusive=True) # Anonymous queue

        # Bind to all lead messages
        await queue.bind(exchange, routing_key='lead.*')

        async with queue.iterator() as queue_iter:
            async for message in queue_iter:
                await self.process_lead(message)
                await message.ack()
```

Consumer: Notifier Service

```
# notifier - слушает системные события от всех сервисов
class EventConsumer:
    async def consume_events(self):
        exchange = await channel.declare_exchange('events_exchange', ExchangeType.TOPIC)
        queue = await channel.declare_queue('notifier_queue')

        # Bind to error events
        await queue.bind(exchange, routing_key='*.error')
        await queue.bind(exchange, routing_key='*.crash')
```

4. Реальные события и типы сообщений

События Scripts Mode

```
@dataclass
class NewActiveScript(BaseEvent):
    script_for_campaign_id: str

class ScriptStartedEvent(Event):
    sfc_id: UUID
    chats: list[str | int]

class ScriptFinishedEvent(Event):
    sfc_id: UUID
    finished_at: datetime
    problems: list[Optional[UUID]]
```

События Parser Mode

```
@dataclass(kw_only=True)
class NewTargetMessage(BaseEvent):
    worker_id: str
    campaign_id: str
    chat_id: int
    username: str
    message: str
```

Системные события

```
class ServiceCrashedEvent(Event):
    service: Service
    reason: str

class BotBannedEvent(Event):
    worker_id: UUID
    comment: str
```

5. MQ архитектура и routing

Exchange типы и routing keys

- scripts_exchange (Topic): admin.script_created
- leads_exchange (Topic): lead.{campaign_id} , lead.*
- events_exchange (Topic): {service}.error , {service}.crash , {service}.started

Durable queues

```
# Все очереди durable для persistence
queues:
    scripts_queue:
        durable: true
        dead-letter-exchange: scripts_dlx

    leads_queue:
        durable: true
        dead-letter-exchange: leads_dlx
```

Message acknowledgment

```
# Consumer обрабатывает сообщение
try:
    await process_message(message)
    await message.ack() # Успешно обработано
except Exception as e:
    await message.nack(requeue=False) # Отправить в DLQ
```

6. Масштабирование и отказоустойчивость

Consumer pools

```
# scripts/infrastructure/mq/consumer.py
connection_pool: Pool = Pool(self._get_connection, max_size=2)
channel_pool: Pool = Pool(self._get_channel, max_size=10)
```

Dead Letter Queues

```
# Обработка неуспешных сообщений
dead-letter-queues:
    scripts_dlx:
        routing_key: scripts.failed
        ttl: 86400000 # 24 часа
```

```
leads_dlx:
  routing_key: leads.failed
  ttl: 3600000    # 1 час
```

Load balancing между consumers

Scripts workers: Каждый consumer получает уникальный script_id

Manager instances: Round-robin распределение лидов

Notifier: Получает все системные события

7. Мониторинг очередей

RabbitMQ Management API

```
# Получение статистики очередей
async def get_queue_stats():
    # messages, consumers, publish rate, etc.
    stats = await management_api.get_queue_info('scripts_queue')
    return stats
```

Метрики Prometheus

```
# MQ метрики
rabbitmq_queue_messages{queue="scripts_queue"}
rabbitmq_queue_consumers{queue="leads_queue"}
rabbitmq_publish_rate{service="admin"}
rabbitmq_consume_rate{service="scripts"}
```

Вывод

Проект CoolThing демонстрирует эффективное использование очередей сообщений для реализации сложной распределенной системы:

- Два режима работы с разными паттернами взаимодействия
- Topic exchanges с гибким routing'ом сообщений
- Publisher-Subscriber для координации выполнения скриптов
- Event-Driven для обработки лидов и системных событий
- Отказоустойчивость через DLQ и acknowledgments
- Масштабируемость через consumer pools и load balancing

Вывод по работе

Межсервисное взаимодействие включает:

- RabbitMQ как центральная шина сообщений
- Topic exchanges с routing keys по типам сообщений
- Durable queues для надежности доставки
- Dead letter queues для обработки ошибок
- Consumer pools для масштабирования
- Event-driven архитектура для loosely coupled сервисов