Лабораторная работа №6. Настройка межсервисного взаимодействия (RabbitMQ)

1. Введение

В проекте **BusinessThing** реализовано асинхронное межсервисное взаимодействие через брокер сообщений **RabbitMQ**. Подход используется для развязки сервисов и выполнения «тяжёлых» фоновых задач без блокировки пользовательских запросов.

Реализован сценарий обработки документов:

- **Producer:** `core-service` — создаёт/обновляет сущности документов и публикует задания на обработку.
- **Consumer:** `docs-processor` — воркер, который забирает задания из очереди и выполняет обработку.

2. Настройка инфраструктуры

RabbitMQ разворачивается в составе общего окружения через Docker Compose.

Файл: `BusinessThing/rabbitmq/compose.yaml`

```
services:
  rabbitmq:
    image: rabbitmq:3-management-alpine
    profiles: [dev, full]
    container_name: rabbitmq
    ports:
      - "5672:5672"
      - "15672:15672"
    volumes:
      - rabbitmq-data:/var/lib/rabbitmq
    restart: unless-stopped
    healthcheck:
      test: ["CMD", "rabbitmq-diagnostics", "status"]
      interval: 30s
      timeout: 10s
      retries: 5

volumes:
  rabbitmq-data:
    name: rabbitmq-data
```

RabbitMQ подключается в общий `compose.yaml` через `include` (см. `BusinessThing/compose.yaml`), что позволяет поднимать всю систему одной командой.

3. Реализация клиента очереди (shared approach)

В обоих сервисах используется схожая обёртка над `amqp091-go`:

- устанавливается соединение (`amqp.Dial`);
- открывается канал (`conn.Channel()`);
- объявляется durable очередь (`QueueDeclare(queueName, durable=true, ...)`);
- сообщения публикуются как `application/json` и помечаются persistent.

3.1 Producer (core-service)

Файл: `BusinessThing/core-service/internal/queue/rabbitmq_client.go`

Ключевые моменты:

- очередь объявляется как durable;
- публикация происходит через `PublishWithContext`;
- `DeliveryMode: amqp.Persistent`.

```
func (c *RabbitMQClient) PublishMessage(ctx context.Context, message
interface{}) error {
    span, ctx := opentracing.StartSpanFromContext(ctx,
"queue.RabbitMQClient.PublishMessage")
    defer span.Finish()

    body, err := json.Marshal(message)
    if err != nil {
        return fmt.Errorf("failed to marshal message: %w", err)
    }

    return c.channel.PublishWithContext(
        ctx,
        "",
        c.queueName,
        false,
        false,
        amqp.Publishing{
            ContentType:  "application/json",
            Body:         body,
            DeliveryMode: amqp.Persistent,
        },
    )
}
```

Публикация заданий используется, например, в сервисах доменной логики `core-service/internal/service/...`.

3.2 Consumer (docs-processor)

Файл: `BusinessThing/docs-processor/internal/queue/rabbitmq_client.go`

Воркер настраивает QoS (prefetch), подписывается на очередь с ручным подтверждением (`autoAck=false`) и обрабатывает сообщения в цикле.

Ключевые аспекты обработки:

- `Ack` при успешной обработке;
- `Nack(false, false)` при ошибке (без re-queue на уровне брокера);
- retry реализован прикладным образом: при ошибке, если можно повторить — публикуется новая задача с увеличенным `RetryCount`.

```
func (c *RabbitMQClient) ConsumeJobs(ctx context.Context, consumerTag
string, prefetchCount int, handler JobHandler) error {
    if err := c.channel.Qos(prefetchCount, 0, false); err != nil {
        return fmt.Errorf("failed to set QoS: %w", err)
    }

    msgs, err := c.channel.Consume(
        c.queueName,
        consumerTag,
        false,
        false,
        false,
        false,
        nil,
    )
    if err != nil {
        return fmt.Errorf("failed to register consumer: %w", err)
    }

    for {
        select {
        case <-ctx.Done():
            return nil
        case msg, ok := <-msgs:
            if !ok {
                return nil
            }
            c.handleMessage(ctx, msg, handler)
        }
    }
}
```

4. Наблюдаемость и надёжность

- В клиенте публикации/обработки используется tracing через `opentracing.StartSpanFromContext`.
- Очередь настроена как durable, сообщения публикуются persistent.
- Consumer использует ручные подтверждения и управляет повторными попытками на уровне приложения.

5. Вывод

Настроено межсервисное взаимодействие через RabbitMQ по модели producer/consumer: `core-service` публикует задания на обработку документов, `docs-processor` асинхронно их выполняет. Решение снижает связность сервисов и позволяет выполнять ресурсоёмкие операции в фоне.