Домашняя работа №2. Реализация межсервисного взаимодействия (RabbitMQ)

1. Введение

В проекте `BusinessThing` реализовано асинхронное межсервисное взаимодействие с использованием брокера сообщений **RabbitMQ**. Данный подход используется для развязки сервисов и обработки тяжелых задач, таких как процессинг документов.

Взаимодействие происходит между:

- **Producer**: `core-service` (Основной сервис, принимающий документы)
- **Consumer**: `docs-processor` (Воркер, занимающийся чисткой, чанпингом и эмбеддингом документов)

2. Настройка инфраструктуры

Для развертывания RabbitMQ используется Docker Compose.

Файл: `rabbitmq/compose.yaml` (подключается в основном `compose.yaml`)

```
services:
  rabbitmq:
    image: rabbitmq:3.12-management-alpine
    restart: unless-stopped
    ports:
      - "${RABBITMQ_PORT:-5672}:5672"
      - "${RABBITMQ_UI_PORT:-15672}:15672"
    environment:
      RABBITMQ_DEFAULT_USER: ${RABBITMQ_USER:-guest}
      RABBITMQ_DEFAULT_PASS: ${RABBITMQ_PASSWORD:-guest}
    healthcheck:
      test: ["CMD", "rabbitmqctl", "status"]
      interval: 10s
      timeout: 5s
      retries: 5
    networks:
      - business_thing_network
```

3. Реализация клиента (Shared Code)

Оба сервиса используют схожую реализацию обертки над `amqp091-go`.

Пример инициализации соединения и канала (`core-service/internal/queue/rabbitmq_client.go`):

```
func NewRabbitMQClient(url, queueName string) (*RabbitMQClient, error) {
    conn, err := amqp.Dial(url)
    if err != nil {
        return nil, fmt.Errorf("failed to connect to RabbitMQ: %w", err)
    }

    channel, err := conn.Channel()
    if err != nil {
        conn.Close()
        return nil, fmt.Errorf("failed to open channel: %w", err)
    }

    // Объявление durable очереди
    _, err = channel.QueueDeclare(
        queueName,
        true,  // durable
        false, // autoDelete
        false, // exclusive
        false, // noWait
        nil,   // args
    )
    // ...
    return &RabbitMQClient{conn: conn, channel: channel, queueName:
queueName}, nil
}
```

4. Отправка сообщений (Producer)

`core-service` публикует задачи на обработку документов.

Код публикации (`core-service/internal/queue/rabbitmq_client.go`):

```
func (c *RabbitMQClient) PublishMessage(ctx context.Context, message
interface{}) error {
    // ... tracing ...
    body, err := json.Marshal(message)
    // ...
    err = c.channel.PublishWithContext(
        ctx,
        "",           // exchange
        c.queueName,  // routing key
        false,        // mandatory
        false,        // immediate
        amqp.Publishing{
            ContentType:  "application/json",
            Body:         body,
            DeliveryMode: amqp.Persistent, // Сообщения сохраняются на
диске
        },
    )
```

```
    // ...
}
```

Использование в сервисе (`core-service/cmd/core-service/main.go`):

```
// Инициализация
queueClient, err := queue.NewRabbitMQClient(cfg.GetRabbitMQURL(),
cfg.GetRabbitMQQueueName())
// Передача в сервис документов
docService := document.New(repo, queueClient, "document_processing")
```

5. Получение сообщений (Consumer)

`docs-processor` подписывается на очередь и обрабатывает входящие задачи.

Код потребления (`docs-processor/internal/queue/rabbitmq_client.go`):

```
func (c *RabbitMQClient) ConsumeJobs(ctx context.Context, consumerTag
string, prefetchCount int, handler JobHandler) error {
    // QoS для равномерного распределения нагрузки
    err := c.channel.Qos(prefetchCount, 0, false)

    msgs, err := c.channel.Consume(
        c.queueName,
        consumerTag,
        false, // autoAck = false (ручное подтверждение)
        false,
        false,
        false,
        nil,
    )

    go func() {
        for d := range msgs {
            // Декодирование задачи
            var job domain.ProcessingJob
            json.Unmarshal(d.Body, &job)

            // Вызов бизнес-логики
            if err := handler(ctx, &job); err != nil {
                // В случае ошибки возвращаем в очередь (nack)
                d.Nack(false, true)
            } else {
                // Подтверждение успешной обработки (ack)
                d.Ack(false)
            }
        }
    }()
```

```
    return nil
}
```

Запуск воркера (`docs-processor/cmd/worker/main.go`):

```
if err := rabbitMQ.ConsumeJobs(ctx, "docs-processor-worker", 1,
service.ProcessDocument); err != nil {
    logger.Fatal(ctx, "Failed to start consumer", "error", err)
}
```

6. Вывод

Реализована надежная схема доставки сообщений с `Durable` очередями и `Persistent` сообщениями. Используется механизм `Manual Ack` для гарантии обработки сообщений: сообщение удаляется из очереди только после успешного выполнения бизнес-логики воркером.