**Лабораторная работа №5. Разработка микросервисной архитектуры на .NET с использованием RabbitMQ и Kafka**

Целью данной лабораторной работы является изучение и практическая реализация принципов микросервисной архитектуры на платформе .NET с применением технологий асинхронной коммуникации между сервисами через брокеры сообщений RabbitMQ и Apache Kafka, что позволит освоить механизмы организации распределенных систем, научиться проектировать автономные сервисы, реализовывать взаимодействие между ними на основе паттернов обмена сообщениями, а также сравнить особенности и применимость каждой технологии в зависимости от специфики задачи и требований к производительности, масштабируемости и отказоустойчивости системы.

**Теоретическая часть:**

**1. Введение в микросервисную архитектуру**

Микросервисная архитектура — это подход к проектированию и разработке программного обеспечения, при котором приложение разделяется на набор небольших, автономных сервисов, каждый из которых выполняет конкретную бизнес-функцию и может развиваться, развертываться и поддерживаться независимо от других сервисов.

**Реализация микросервисов в .NET**

**1.1. Создание проекта микросервиса**

В Visual Studio создайте новый проект типа "ASP.NET Core Web API". Это будет базовый шаблон для нашего микросервиса.

**1.2. Структура проекта**

Разобьем проект на следующие основные части:

* **Controllers**: Обработка входящих запросов.
* **Services**: Бизнес-логика.
* **Repositories**: Доступ к данным.
* **Models**: Определение структур данных.

|  |
| --- |
| OrderService/  ├── Controllers/  │ └── OrdersController.cs  ├── Services/  │ └── OrderService.cs  ├── Repositories/  │ └── OrderRepository.cs  ├── Models/  │ └── Order.cs  └── Program.cs |

**1.3. Определение модели данных**

Создадим модель **Order** в папке **Models**.

|  |
| --- |
| namespace OrderService.Models  {  public class Order  {  public int Id { get; set; }  public string CustomerName { get; set; }  public decimal TotalAmount { get; set; }  public DateTime CreatedAt { get; set; }  }  } |

**1.4. Реализация репозитория**

Создадим класс **OrderRepository** для работы с данными. Для упрощения используем встроенный список вместо базы данных.

|  |
| --- |
| using System.Collections.Generic;  using OrderService.Models;  namespace OrderService.Repositories  {  public class OrderRepository  {  private static List<Order> \_orders = new List<Order>();  public IEnumerable<Order> GetAllOrders()  {  return \_orders;  }  public Order GetOrderById(int id)  {  return \_orders.Find(o => o.Id == id);  }  public void AddOrder(Order order)  {  order.Id = \_orders.Count + 1;  order.CreatedAt = DateTime.UtcNow;  \_orders.Add(order);  }  }  } |

**1.5. Реализация сервиса**

Создадим сервис **OrderService**, который будет использовать репозиторий для выполнения бизнес-логики.

|  |
| --- |
| using System.Collections.Generic;  using OrderService.Models;  using OrderService.Repositories;  namespace OrderService.Services  {  public class OrderService  {  private readonly OrderRepository \_repository;  public OrderService(OrderRepository repository)  {  \_repository = repository;  }  public IEnumerable<Order> GetAllOrders()  {  return \_repository.GetAllOrders();  }  public Order GetOrderById(int id)  {  return \_repository.GetOrderById(id);  }  public void CreateOrder(Order order)  {  \_repository.AddOrder(order);  }  }  } |

**1.6. Настройка контроллера**

Создадим контроллер **OrdersController** для обработки HTTP-запросов.

|  |
| --- |
| using Microsoft.AspNetCore.Mvc;  using System.Collections.Generic;  using OrderService.Models;  using OrderService.Services;  namespace OrderService.Controllers  {  [ApiController]  [Route("api/[controller]")]  public class OrdersController : ControllerBase  {  private readonly OrderService \_orderService;  public OrdersController(OrderService orderService)  {  \_orderService = orderService;  }  [HttpGet]  public ActionResult<IEnumerable<Order>> Get()  {  var orders = \_orderService.GetAllOrders();  return Ok(orders);  }  [HttpGet("{id}")]  public ActionResult<Order> Get(int id)  {  var order = \_orderService.GetOrderById(id);  if (order == null)  return NotFound();  return Ok(order);  }  [HttpPost]  public ActionResult<Order> Post([FromBody] Order order)  {  \_orderService.CreateOrder(order);  return CreatedAtAction(nameof(Get), new { id = order.Id }, order);  }  }  } |

**1.7. Настройка зависимостей**

Добавим зависимости в **Program.cs** или **Startup.cs** (в зависимости от версии .NET).

|  |
| --- |
| var builder = WebApplication.CreateBuilder(args);  // Регистрация зависимостей  builder.Services.AddSingleton<OrderRepository>();  builder.Services.AddScoped<OrderService>();  var app = builder.Build();  app.MapControllers();  app.Run(); |

**1.8.Запуск и тестирование**

Запустите проект и протестируйте его с помощью **Postman** или **curl**.

**POST запрос для создания заказа:**

|  |
| --- |
| curl -X POST http://localhost:5000/api/orders \  -H "Content-Type: application/json" \  -d '{"CustomerName": "Denis Bondarenko", "TotalAmount": 100}' |

**GET запрос для получения списка заказов:**

|  |
| --- |
| curl http://localhost:5000/api/orders |

**2. Ключевые характеристики микросервисной архитектуры (!)**

1. **Автономность**:
   * Каждый микросервис является самостоятельным компонентом, который имеет свою базу данных, собственную кодовую базу и может быть развернут независимо от других сервисов.
   * Например, в интернет-магазине сервис "каталог товаров" не зависит напрямую от сервиса "система заказов".
2. **Специализация**:
   * Каждый микросервис решает одну конкретную задачу или реализует один бизнес-процесс. Это позволяет упростить разработку и тестирование каждого компонента.
   * Пример: сервис "обработка платежей" занимается только взаимодействием с платежными шлюзами.
3. **Децентрализованное управление данными**:
   * Каждый микросервис использует собственную базу данных, что исключает зависимость от общей базы данных для всех сервисов.
   * Например, сервис "корзина покупок" может использовать реляционную базу данных (SQL), а сервис "аналитика" — NoSQL базу данных для хранения событий.
4. **Гибкость технологий**:
   * Разные микросервисы могут использовать различные технологии, языки программирования, фреймворки и инструменты в зависимости от требований задачи.
   * Например, один сервис может быть написан на C# (.NET), другой — на Java, а третий — на Python.
5. **Асинхронная коммуникация**:
   * Микросервисы часто общаются друг с другом через асинхронные механизмы, такие как очереди сообщений (RabbitMQ, Kafka) или REST API.
   * Асинхронность позволяет системе оставаться устойчивой даже при временных сбоях в одном из сервисов.
6. **Независимое развертывание**:
   * Каждый микросервис можно развернуть отдельно, без необходимости перезапуска всей системы. Это упрощает процесс внедрения новых функций и исправления ошибок.
7. **Масштабируемость**:
   * Поскольку каждый микросервис является независимым, его можно масштабировать горизонтально (увеличение количества экземпляров) или вертикально (увеличение мощности сервера) в зависимости от нагрузки.

**2. Коммуникация между микросервисами**

Для обмена данными между микросервисами можно использовать различные подходы:

* HTTP/REST API: Прямые вызовы между сервисами через HTTP.
* Message Broker: Асинхронное взаимодействие через брокеры сообщений, такие как RabbitMQ или Apache Kafka.

В рамках данной лабораторной работы мы рассмотрим использование RabbitMQ и Kafka для организации коммуникации между микросервисами.

**3.** **RabbitMQ: Очередь сообщений для асинхронной обработки**

RabbitMQ — это популярный брокер сообщений, который позволяет реализовать паттерн "publisher-subscriber". В интернет-магазине RabbitMQ может использоваться для таких задач, как отправка уведомлений о новых заказах или обновление статуса товара.

**3.1. Настройка RabbitMQ**

Для начала установите RabbitMQ на локальном компьютере или используйте облачные решения.

**1. Установка Erlang**

RabbitMQ написан на языке программирования Erlang, поэтому для его работы необходим Erlang runtime. Перед установкой RabbitMQ убедитесь, что Erlang уже установлен на вашем компьютере.

**Для Windows:**

1. Скачайте установщик Erlang с официального сайта:
   * <https://www.erlang.org/downloads>
2. Запустите скачанный установщик **.exe** и следуйте инструкциям мастера установки.
3. После установки проверьте версию Erlang, открыв командную строку (**cmd**) и введя:

|  |
| --- |
| erl -version |

Если команда выполняется успешно, значит Erlang установлен корректно.

**Для Linux (Ubuntu/Debian):**

1. Обновите список пакетов:

|  |
| --- |
| sudo apt update |

2. Установите Erlang:

|  |
| --- |
| sudo apt install erlang |

3. Проверьте версию:

|  |
| --- |
| erl -version |

**2. Установка RabbitMQ**

После установки Erlang можно приступить к установке самого RabbitMQ.

**Для Windows:**

1. Скачайте установщик RabbitMQ с официального сайта:
   * <https://www.rabbitmq.com/download.html>
2. Выберите версию, соответствующую вашей операционной системе, и скачайте установочный файл **.exe**.
3. Запустите установщик и следуйте инструкциям мастера установки.
4. После установки добавьте путь к RabbitMQ в переменную среды **PATH**. Например:

|  |
| --- |
| C:\Program Files\RabbitMQ Server\rabbitmq\_server-<version>\sbin |

5. Проверьте успешность установки, открыв командную строку и выполнив:

|  |
| --- |
| rabbitmqctl status |

**Для Linux (Ubuntu/Debian):**

1. Добавьте репозиторий RabbitMQ:

|  |
| --- |
| sudo apt update  sudo apt install curl gnupg  curl -fsSL https://github.com/rabbitmq/signing-keys/releases/download/2.0/rabbitmq-release-signing-key.asc | sudo apt-key add -  echo "deb https://dl.bintray.com/rabbitmq-erlang/debian $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/bintray.erlang.list  echo "deb https://dl.bintray.com/rabbitmq/debian $(lsb\_release -cs) main" | sudo tee /etc/apt/sources.list.d/bintray.rabbitmq.list |

2. Обновите список пакетов:

|  |
| --- |
| sudo apt update |

3. Установите RabbitMQ:

|  |
| --- |
| sudo apt install rabbitmq-server |

4. Запустите службу RabbitMQ:

|  |
| --- |
| sudo systemctl start rabbitmq-server |

5. Включите автозапуск при загрузке системы:

|  |
| --- |
| sudo systemctl enable rabbitmq-server |

6. Проверьте статус:

|  |
| --- |
| sudo rabbitmqctl status |

**3. Настройка Web-интерфейса RabbitMQ Management Plugin**

RabbitMQ предоставляет удобный веб-интерфейс для управления очередями и сообщениями. Чтобы его использовать, нужно активировать плагин **rabbitmq\_management**.

**Активация плагина:**

1. Откройте терминал или командную строку.
2. Выполните команду:

|  |
| --- |
| rabbitmq-plugins enable rabbitmq\_management |

3. После активации плагина перейдите в браузере по адресу:

|  |
| --- |
| http://localhost:15672 |

4. По умолчанию имя пользователя и пароль — **guest:guest**.

**3.2. Пример использования RabbitMQ в интернет-магазине**

Создание очереди и отправка сообщения

|  |
| --- |
| using RabbitMQ.Client;  using System.Text;  // Подключение к RabbitMQ  var factory = new ConnectionFactory() { HostName = "localhost" };  using var connection = factory.CreateConnection();  using var channel = connection.CreateModel();  // Создание очереди  channel.QueueDeclare(queue: "order\_queue", durable: false, exclusive: false, autoDelete: false, arguments: null);  // Отправка сообщения  string message = "Новый заказ создан!";  var body = Encoding.UTF8.GetBytes(message);  channel.BasicPublish(exchange: "", routingKey: "order\_queue", basicProperties: null, body: body);  Console.WriteLine(" [x] Sent {0}", message); |

Получение сообщения

|  |
| --- |
| using RabbitMQ.Client;  using RabbitMQ.Client.Events;  using System.Text;  var factory = new ConnectionFactory() { HostName = "localhost" };  using var connection = factory.CreateConnection();  using var channel = connection.CreateModel();  // Подключение к очереди  channel.QueueDeclare(queue: "order\_queue", durable: false, exclusive: false, autoDelete: false, arguments: null);  var consumer = new EventingBasicConsumer(channel);  consumer.Received += (model, ea) =>  {  var body = ea.Body.ToArray();  var message = Encoding.UTF8.GetString(body);  Console.WriteLine(" [x] Received {0}", message);  };  channel.BasicConsume(queue: "order\_queue", autoAck: true, consumer: consumer);  Console.WriteLine(" Press [enter] to exit.");  Console.ReadLine(); |

**3.2 Применение RabbitMQ в интернет-магазине**

* Когда пользователь создает новый заказ, сервис "Система заказов" отправляет сообщение в очередь **order\_queue**.
* Сервис "Уведомления" подписывается на эту очередь и отправляет email или SMS клиенту.

**4. Apache Kafka: Распределенная система потоковой передачи данных**

Apache Kafka — это распределенная система потоковой передачи данных, которая подходит для обработки больших объемов данных в реальном времени. В интернет-магазине Kafka может использоваться для таких задач, как анализ поведения пользователей или мониторинг производительности системы.

**4.1. Настройка Kafka**

**1. Установка Kafka на Windows**

**Шаг 1: Установка JDK**

Kafka работает на платформе Java, поэтому необходимо установить JDK:

1. Скачайте последнюю версию JDK с официального сайта [Oracle](https://www.oracle.com/java/technologies/javase-downloads.html) или [OpenJDK](https://openjdk.org/) .
2. Установите JDK, следуя инструкциям установщика.

**Шаг 2: Установка Apache Kafka**

1. Скачайте Kafka с официального сайта: <https://kafka.apache.org/downloads>.
2. Выберите архив (например, kafka\_2.13-3.4.0.tgz).
3. Распакуйте архив в желаемую директорию (например, C:\kafka).

**Шаг 3: Настройка переменных среды**

1. Откройте "Системные свойства" → "Дополнительные параметры системы" → "Переменные среды".
2. Добавьте путь к бинарным файлам Kafka (C:\kafka\bin\windows) в переменную Path.

**Шаг 4: Запуск Zookeeper и Kafka**

Kafka использует Zookeeper для управления конфигурацией и координацией узлов.

1. Откройте две командные строки (одна для Zookeeper, другая для Kafka).
2. В первой командной строке запустите Zookeeper:

|  |
| --- |
| kafka-server-start.bat ..\config\zookeeper.properties |

3. Во второй командной строке запустите Kafka:

|  |
| --- |
| kafka-server-start.bat ..\config\server.properties |

**Шаг 5: Создание топика**

1. Создайте новый топик, например **test-topic**:

|  |
| --- |
| kafka-topics.bat --create --topic test-topic --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1 |

2. Проверьте список топиков:

|  |
| --- |
| kafka-topics.bat --list --bootstrap-server localhost:9092 |

**4.2. Пример использования Kafka в интернет-магазине**

Создание топика и отправка сообщения

|  |
| --- |
| using Confluent.Kafka;  using System.Text;  var config = new ProducerConfig { BootstrapServers = "localhost:9092" };  // Создание производителя  using var producer = new ProducerBuilder<Null, string>(config).Build();  // Отправка сообщения  var message = "Пользователь добавил товар в корзину";  producer.Produce("user\_events", new Message<Null, string> { Value = message }, deliveryReport =>  {  if (deliveryReport.Error.Code != ErrorCode.NoError)  {  Console.WriteLine($"Ошибка отправки сообщения: {deliveryReport.Error.Reason}");  }  else  {  Console.WriteLine($"Сообщение успешно отправлено: {message}");  }  });  producer.Flush(); |

Получение сообщения

|  |
| --- |
| using Confluent.Kafka;  using System;  var config = new ConsumerConfig  {  GroupId = "user-event-group",  BootstrapServers = "localhost:9092",  AutoOffsetReset = AutoOffsetReset.Earliest  };  // Создание потребителя  using var consumer = new ConsumerBuilder<Ignore, string>(config).Build();  consumer.Subscribe("user\_events");  try  {  while (true)  {  var consumeResult = consumer.Consume();  Console.WriteLine($"Получено сообщение: {consumeResult.Message.Value}");  }  }  catch (OperationCanceledException)  {  consumer.Close();  } |

Практические задания:

**Задание 1: Разработка микросервиса "Создание заказа"**

**Цель:**

Реализовать микросервис, который обрабатывает создание заказов и отправляет уведомления о новых заказах в очередь RabbitMQ.

**Требования:**

1. Создайте микросервис "Система заказов" (**OrderService**).
2. При создании нового заказа:
   * Сохраните информацию о заказе в базу данных.
   * Отправьте сообщение в очередь RabbitMQ (**order\_created\_queue**) с информацией о новом заказе.
3. Используйте Entity Framework для работы с базой данных.

**Задание 2: Реализация микросервиса "Уведомления"**

**Цель:**

Создать микросервис, который получает уведомления о новых заказах из очереди RabbitMQ и отправляет email/SMS клиентам.

**Требования:**

1. Создайте микросервис "Уведомления" (**NotificationService**).
2. Подпишитесь на очередь **order\_created\_queue** в RabbitMQ.
3. При получении сообщения:

* Извлеките данные о заказе.
* Сымитируйте отправку email или SMS клиенту (например, выведите сообщение в консоль).

**Задание 3: Анализ поведения пользователей с помощью Kafka**

**Цель:**

Реализовать систему сбора данных о действиях пользователей (например, добавление товаров в корзину) с использованием Kafka.

**Требования:**

1. Создайте топик Kafka **user\_events**.
2. В микросервисе "Корзина покупок" (**CartService**):
3. При добавлении товара в корзину отправляйте событие в топик **user\_events**.
4. В микросервисе "Аналитика" (**AnalyticsService**):

* Подпишитесь на топик **user\_events**.
* Собирайте статистику о количестве добавленных товаров за определенный период времени.

**Задание 4: Обновление статуса заказа через RabbitMQ**

**Цель:**

Реализовать механизм обновления статуса заказа с использованием RabbitMQ.

**Требования:**

1. Создайте очередь RabbitMQ **order\_status\_updates**.
2. В микросервисе "Система заказов":

* Когда статус заказа изменяется (например, "Ожидание оплаты" → "Оплачен"), отправьте сообщение в очередь **order\_status\_updates**.

1. В микросервисе "Уведомления":

* Подпишитесь на очередь **order\_status\_updates**.
* Отправьте уведомление клиенту об изменении статуса заказа.

**Задание 5: Мониторинг производительности системы с помощью Kafka**

**Цель:**

Создать систему мониторинга, которая собирает метрики о работе микросервисов и отправляет их в Kafka.

**Требования:**

1. Создайте топик **Kafka system\_metrics**.

2. В каждом микросервисе:

* Собирайте метрики (например, время выполнения запросов, количество обработанных заказов).
* Периодически отправляйте метрики в топик **system\_metrics**.

В микросервисе "Мониторинг" (MonitoringService):

* Подпишитесь на топик **system\_metrics**.
* Выводите собранные метрики в консоль или сохраняйте их в базу данных.