.NET Microservices – Azure DevOps and AKS

Section 11: Asynchronous Microservice Communication using RabbitMQ- Notes

Introduction to RabbitMQ

RabbitMQ is an open-source message broker that enables applications to communicate with each other by sending messages. It implements the Advanced Message Queuing Protocol (AMQP) for message queuing. RabbitMQ is widely used in microservices architectures to enable asynchronous communication between services.

Key Concepts:

- **Broker**: RabbitMQ server that manages queues and exchanges.
- **Queue**: Storage for messages until they are processed.
- **Exchange**: Routes messages to one or more queues based on routing rules.
- **Binding**: The relationship between an exchange and a queue that determines how messages are routed.
- **Producer**: Application that sends messages to RabbitMQ.
- **Consumer**: Application that receives messages from RabbitMQ.

Types of RabbitMQ Exchanges

RabbitMQ supports several types of exchanges, each serving different routing needs:

1. **Direct Exchange**:
    - **Purpose**: Routes messages to queues based on a routing key.
    - **Use Case**: Direct routing of messages to a specific queue. Ideal for scenarios where you want to route messages with a specific routing key to a particular queue.
    - **Example**: A logging system where logs of different severity levels (info, error) are routed to different queues.

```
{
  "exchangeType": "direct",
  "routingKey": "info",
  "queueName": "info-logs"
}
```

2. **Fanout Exchange**:
    - **Purpose**: Routes messages to all queues bound to it, ignoring the routing key.
    - **Use Case**: Broadcasting messages to multiple queues. Suitable for scenarios where you want all subscribers to receive the same message.
    - **Example**: A notification system where all notifications are broadcasted to multiple subscribers.

```
{
 "exchangeType": "fanout",
 "queueName": "notifications"
}
```

3. **Topic Exchange**:
    - **Purpose**: Routes messages to queues based on matching between routing key patterns and the binding key patterns.
    - **Use Case**: Routing messages to multiple queues based on complex routing rules. Ideal for scenarios with multiple consumers interested in specific types of messages.
    - **Example**: A news service where messages are routed based on topics like sports, politics, and technology.

```
{
 "exchangeType": "topic",
 "routingKeyPattern": "news.sports",
 "queueName": "sports-news"
}
```

4. **Headers Exchange**:
    - **Purpose**: Routes messages based on message headers instead of routing keys.
    - **Use Case**: Advanced routing based on multiple header attributes. Useful for complex routing scenarios.
    - **Example**: A document processing system where documents are routed based on their type and priority.

```
{
 "exchangeType": "headers",
```

```
  "headers": {
    "type": "invoice",
    "priority": "high"
  },
  "queueName": "high-priority-invoices"
}
```

Creating RabbitMQ Container in Docker Compose File

To run RabbitMQ in a Docker container, you can use Docker Compose. Here's a sample docker-compose.yml file to set up RabbitMQ:

```
version: '3'

services:
  rabbitmq:
    image: "rabbitmq:3-management"
    container_name: rabbitmq
    ports:
      - "5672:5672"   # RabbitMQ default port
      - "15672:15672" # Management UI port
    environment:
      RABBITMQ_DEFAULT_USER: user
      RABBITMQ_DEFAULT_PASS: password
```

Explanation:

- image: Specifies the RabbitMQ Docker image to use. 3-management includes the RabbitMQ Management Plugin for a web-based UI.
- ports: Maps the container ports to host ports. 5672 is the default RabbitMQ port, and 15672 is for the Management UI.
- environment: Sets default credentials for RabbitMQ.

Publishing Messages in a Service Class in Business Logic Layer

To publish messages to RabbitMQ from a service class, you can use the RabbitMQ.Client library. Here's a basic example in a .NET Core service class:

```
using RabbitMQ.Client;
using System.Text;

public class MessagePublisher
{
    private readonly IConnection _connection;
    private readonly IModel _channel;

    public MessagePublisher()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        _connection = factory.CreateConnection();
        _channel = _connection.CreateModel();
    }

    public void PublishMessage(string message, string exchangeName, string routingKey)
    {
        _channel.ExchangeDeclare(exchange: exchangeName, type: ExchangeType.Direct);

        var body = Encoding.UTF8.GetBytes(message);

        _channel.BasicPublish(exchange: exchangeName,
                              routingKey: routingKey,
                              basicProperties: null,
                              body: body);
    }
}
```

Explanation:

- ConnectionFactory: Creates a connection to RabbitMQ.
- IConnection: Represents the RabbitMQ connection.
- IModel: Represents a channel within the connection.
- ExchangeDeclare: Declares the exchange where messages will be sent.
- BasicPublish: Publishes the message to the specified exchange with a routing key.

Consuming Messages in a Service Class in Business Logic Layer

To consume messages from RabbitMQ, use the following example:

```
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;

public class MessageConsumer
{
    private readonly IConnection _connection;
    private readonly IModel _channel;

    public MessageConsumer()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        _connection = factory.CreateConnection();
        _channel = _connection.CreateModel();
    }

    public void ConsumeMessages(string queueName)
    {
        _channel.QueueDeclare(queue: queueName,
                              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($"Received {message}");
        };

        _channel.BasicConsume(queue: queueName,
                autoAck: true,
                consumer: consumer);
    }
}
```

Explanation:

- QueueDeclare: Declares a queue to receive messages.
- EventingBasicConsumer: Handles messages received from RabbitMQ.
- Received: Event triggered when a message is received.
- BasicConsume: Starts consuming messages from the specified queue.

Key Points to Remember (for Interview Preparation)

- **RabbitMQ**: Message broker that supports asynchronous communication between services.
- **Exchanges**: Direct, Fanout, Topic, and Headers exchanges have different routing capabilities.
- **Direct Exchange**: Routes based on exact routing keys.
- **Fanout Exchange**: Routes messages to all bound queues.
- **Topic Exchange**: Routes based on pattern matching of routing keys.
- **Headers Exchange**: Routes based on message headers.
- **Docker Compose**: Used to define and run multi-container Docker applications.

- **Publishing/Consuming**: Use RabbitMQ client libraries to interact with RabbitMQ servers.
- **Management UI**: Accessible via port 15672 for monitoring RabbitMQ.

Hosted Service in ASP.NET Core for Running RabbitMQ Consumers

In ASP.NET Core, you can use a hosted service to run background tasks such as consuming messages from RabbitMQ. A hosted service is a long-running process that can run in the background of your application.

Here's how you can create a hosted service to consume RabbitMQ messages:

1. Implement the Hosted Service:

```
using Microsoft.Extensions.Hosting;
using RabbitMQ.Client;
using RabbitMQ.Client.Events;
using System;
using System.Text;
using System.Threading;
using System.Threading.Tasks;

public class RabbitMQConsumerHostedService : IHostedService
{
    private readonly IConnection _connection;
    private readonly IModel _channel;
    private readonly string _queueName = "myQueue"; // Name of the queue to consume from

    public RabbitMQConsumerHostedService()
    {
        var factory = new ConnectionFactory() { HostName = "localhost" };
        _connection = factory.CreateConnection();
        _channel = _connection.CreateModel();
```

```
    _channel.QueueDeclare(queue: _queueName,
                          durable: false,
                          exclusive: false,
                          autoDelete: false,
                          arguments: null);
}

public Task StartAsync(CancellationToken cancellationToken)
{
    var consumer = new EventingBasicConsumer(_channel);
    consumer.Received += (model, ea) =>
    {
        var body = ea.Body.ToArray();
        var message = Encoding.UTF8.GetString(body);
        // Process the message here
        Console.WriteLine($"Received message: {message}");
    };

    _channel.BasicConsume(queue: _queueName,
                          autoAck: true,
                          consumer: consumer);

    return Task.CompletedTask;
}

public Task StopAsync(CancellationToken cancellationToken)
{
    _channel.Close();
    _connection.Close();
    return Task.CompletedTask;
```

```
    }
}
```

Explanation:

- **IHostedService**: An interface that allows you to run background tasks in ASP.NET Core.
- **StartAsync**: Starts the service. Here, it sets up the consumer to listen for messages.
- **StopAsync**: Stops the service and cleans up resources.

2. Register the Hosted Service in Program.cs:

```
builder.Services.AddHostedService<RabbitMQConsumerHostedService>();
```

Explanation:

- **AddHostedService**: Registers the RabbitMQConsumerHostedService to run as a background service.

Fanout Exchange

Fanout Exchange routes messages to all queues bound to it, without considering the routing key. This is useful for scenarios where you need to broadcast messages to multiple consumers.

Example Scenario:

- A notification service that sends a notification to all registered clients.

Setup Example in .NET:

```
_channel.ExchangeDeclare(exchange: "fanout-exchange", type: ExchangeType.Fanout);
_channel.BasicPublish(exchange: "fanout-exchange",
                      routingKey: "",
                      basicProperties: null,
                      body: Encoding.UTF8.GetBytes("Broadcast message"));
```

Topic Exchange

Topic Exchange allows routing messages to queues based on patterns in the routing key. It is useful for complex routing scenarios where you need to route messages to different queues based on multiple criteria.

Example Scenario:

- A news aggregator service where different news topics (e.g., sports, politics) are routed to different queues.

Setup Example in .NET:

```
_channel.ExchangeDeclare(exchange: "topic-exchange", type: ExchangeType.Topic);

_channel.BasicPublish(exchange: "topic-exchange",

                routingKey: "news.sports",

                basicProperties: null,

                body: Encoding.UTF8.GetBytes("Sports news update"));
```

Headers Exchange

Headers Exchange routes messages based on message headers instead of the routing key. This type of exchange provides advanced routing capabilities.

Example Scenario:

- An order processing system where messages are routed based on header attributes like order priority and type.

Setup Example in .NET:

```
_channel.ExchangeDeclare(exchange: "headers-exchange", type: ExchangeType.Headers);


var properties = _channel.CreateBasicProperties();

properties.Headers = new Dictionary<string, object>

{

   { "order-type", "urgent" },

   { "order-priority", "high" }

};


_channel.BasicPublish(exchange: "headers-exchange",
```

```
routingKey: "",
basicProperties: properties,
body: Encoding.UTF8.GetBytes("Urgent order message"));
```

Key Points to Remember (for Interview Preparation)

- **RabbitMQ**: A message broker that supports asynchronous communication between services.
- **Exchanges**:
  - **Direct Exchange**: Routes messages based on exact routing keys.
  - **Fanout Exchange**: Routes messages to all bound queues.
  - **Topic Exchange**: Routes messages based on patterns in routing keys.
  - **Headers Exchange**: Routes messages based on message headers.
- **Hosted Services**: Used to run background tasks in ASP.NET Core applications, such as consuming messages from RabbitMQ.
- **Consumer Setup**: Implement IHostedService to manage RabbitMQ message consumption.